Simplon programmer's guide

Tutorial and interface documentation for the Simplon API library

Leutron Vision

Simplon programmer's guide: Tutorial and interface documentation for the Simplon API library

Leutron Vision

Revision: 1.0-21557. Leutron Vision documentation set.

Publication date 27 October 2013

Table of Contents

General information

Scope of the manual

This manual provides tutorial documentation for the Simplon library.

Related documents

- Simplon getting started — Simplon package overview, installation instructions and quick start guide.

1. Introduction to Simplon API

Simplon is an application programming library, which provides a unified, user friendly API for easy application building. It is based on current *GenICam* and *GenTL* standards.

The *GenICam* and *GenTL* are the current standards in the vision industry for digital imaging devices, assuring compatibility between HW and SW of different vendors. The direct usage of these standards is a quite complex task, it requires you to learn a lot of things, not directly related with the problem you want to solve. Furthermore, while the GenICam API is strongly object oriented, the GenTL API is a plain C API with a completely different way, how to set and get items.

The Simplon API is designed to free you from unnecessary direct work with standards and lets you concentrate really only on your problem or task solution. It hides the standard implementation details and provides a simple-to-use, unified application programming interface. Besides this, it also provides additional useful functionality, like displaying images, preprocessing, saving to files etc. It is designed as a tool with which you can do simple tasks simply, while there are almost no limitations for creating also complex applications.

One of the aims of Simplon was to avoid inventing a new terminology — wherever it is possible to use the terminology used in the above mentioned standards, it is used. In the next text we will assume that you are not familiar with the *GenICam* and *GenTL* standards, so let's start with some elementary explanations.

Simplon acts as a *GenTL consumer*, this means the hardware is accessed through a GenTL library (so called *Provider*) — this library is to be delivered by a hardware vendor, in case his camera or imaging device is not compatible on lower level. The Simplon set of libraries is delivered also with own GenTL Provider library, designed namely for Leutron Vision hardware.

In other words, Simplon is not only an API for hardware of Leutron Vision, but also can be used for devices of other vendors as well, if the appropriate GenTL library for this hardware is available. In cases the imaging hardware is compatible on lower level, there even might not be needed the GenTL library from the other vendor — for example the gigabit Ethernet cameras are usually compatible on the *GigE-Vision protocol* level and thus can work directly with the Leutron Vision GenTL Provider library.

The *GenICam* standard specifies namely, how the camera (or in general a device) expose its features and how can be controlled. The way how the device features are displayed in the feature tree in the Simplon Explorer is mostly derived from the GenICam standard.

Note that in this manual we often use for simplicity the term camera instead of device, because in most cases you will use it with a real physical camera. However, the *GenICam* standard is not limited only to cameras, in fact the imaging device can be for example a scanner or any other device, which provides images.

1.1. Simplon Explorer

Before you start dealing with the Simplon API, it is a good idea to spend some time playing with the **Simplon Explorer** application. It is a tool for exploring the camera capabilities and testing its functionality. It is written as an application based on the Simplon API and thus whatever functionality you see in Simplon Explorer, you should be able to implement later yourself in your application, after you get acquainted with the Simplon API.

Simplon Explorer is described in detail in the Simplon Explorer in detail in the Simplon getting started manual. Please read this chapter first. In the next chapters we will assume you already know how to work with it and know the general concept of the feature tree.

1.1.1. Getting the source code (quick start)

The Simplon Explorer provides a **Source code generator** as a rapid application development (RAD) tool. This generator is documented in detail in the chapter Section 1.6, “Simplon source code generator” [p. 19]. Here we will give you only a brief info how to generate a project with a source code, compile it an run, just to get a very rough idea how it works.

To do so, make the following steps:

- Power off/on the camera so that it gets to a default state (which is usually a free-run mode). Give the camera time to boot-up.
- Run Simplon Explorer and open this camera, check if the acquisition can be started, stop the acquisition.
- Start *Code Generation Wizard*

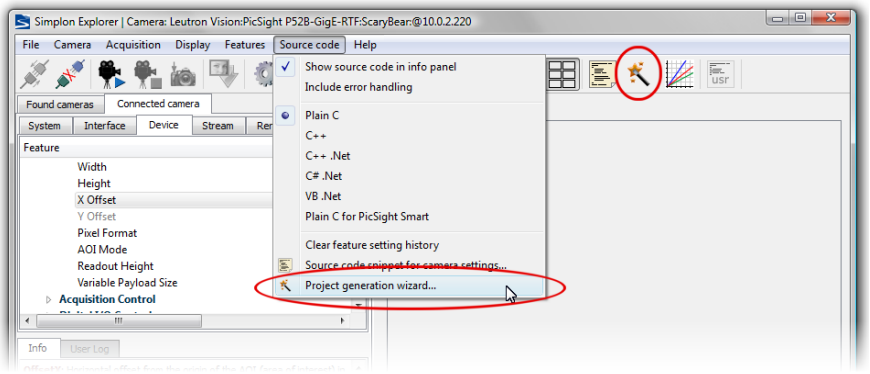


Figure 1.1. Simplon Explorer: Start project generation wizard

- Select the either the “*Simple Windows application, MS Visual Studio*” template, if you are in Windows, or “*Simple Linux Xlib application*” template, if you work with Linux with XWindows.

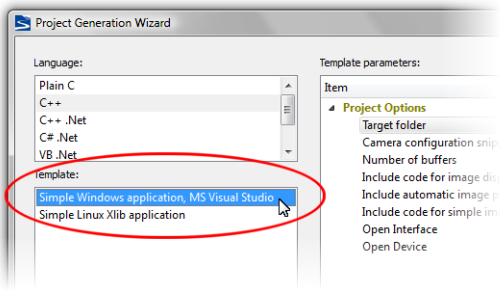


Figure 1.2. Simplon Explorer: Select template

- Select the *target folder*, where the generated project is to be stored.

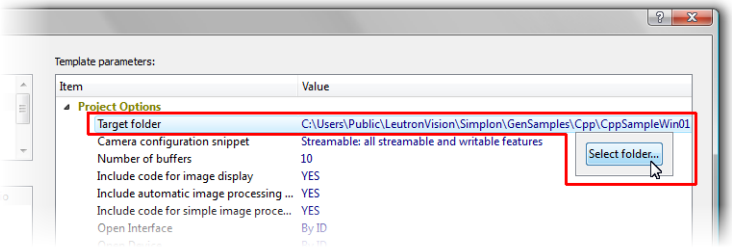


Figure 1.3. Simplon Explorer: Select target folder

- Press the *Generate project* button.
- Close the dialog and disconnect the camera (otherwise it cannot be connected by the generated application).

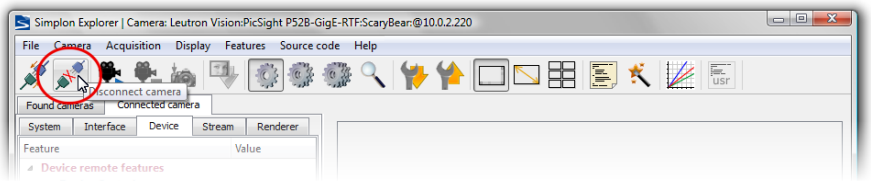


Figure 1.4. Simplon Explorer: Disconnect camera

- Compile the generated project and run it. If everything goes well, you should be able to get the camera working in the generated sample in the same mode as it was working in Simplon Explorer, when the project was generated.

The next chapters will give you necessary introduction needed to understand the generated code and to start adopting the generated code to your needs. Then we get back to the source code generator and describe its functionality in detail in the chapter Section 1.6, “Simplon source code generator” [p. 19].

1.2. Simplon API principles

The aim of Simplon API is a **wide usability**. There exists a lot of imaging software, which can utilize only standard libraries (DLL) with *a plain set of C functions*. On the other hand, *a class oriented API* is very useful, when you have a tools like Intellisense in MS Visual Studio, and even necessary, if you want to use environments like MS .Net Framework.

Another important issue is the **backward compatibility** — nobody wants to be forced to re-compile all his applications just because he/she had received from us updated libraries with necessary bugfixes.

The Simplon library is available with several interfaces:

- A **Plain C API**. This API is provided for the compatibility with non-object oriented languages, also the backward compatibility is assured on this level, freeing you from necessity to recompile your applications with new versions of Simplon.
- A **C++ class wrapper** around the C API: The purpose of the wrapper is namely more comfortable programming. This wrapper consists of one cpp unit with source code, which should be included and compiled with your application.

- A **.Net Class Library** wrapper, which provides set of managed classes, with almost identical API as the C++ class wrapper. This wrapper is provided in single DLL and should be distributed with your application.

The C++ classes are just wrappers around the plain C API. This might seem to be ineffective, but actually it is not — and it brings a big advantage: the backward compatibility is kept on the plain C API, so even if the C++ class library is directly compiled with your application, or .Net Class Library version distributed with your application, you can still update the Simplon libraries to newer version without need to recompile your applications. This is an important aspect in maintenance and troubleshooting.

The existence of three APIs makes more difficult to document all of them in one manual. On the other hand, once you see the code in one API, it is usually quite easy to convert it to another API, because the naming conventions are kept as close as possible. So for example a call in the plain C API looks like this:

```
LvSetInt(hDevice, LvDevice_Width, 1024);
```

And the same code in C++ API:

```
pDevice->SetInt(LvDevice_Width, 1024);
```

In this Simplon API User's Guide we use the C++ API, in the Reference Guide you will then find all of them.

1.3. Simplon Architecture

The figure shows the Simplon main modules (the names beginning with Lv are presented as classes in the C++ API and handles in the plain C API): `LvSystem`, `LvInterface`, `LvDevice`, `LvStream`, `LvEvent`, `LvRenderer`.

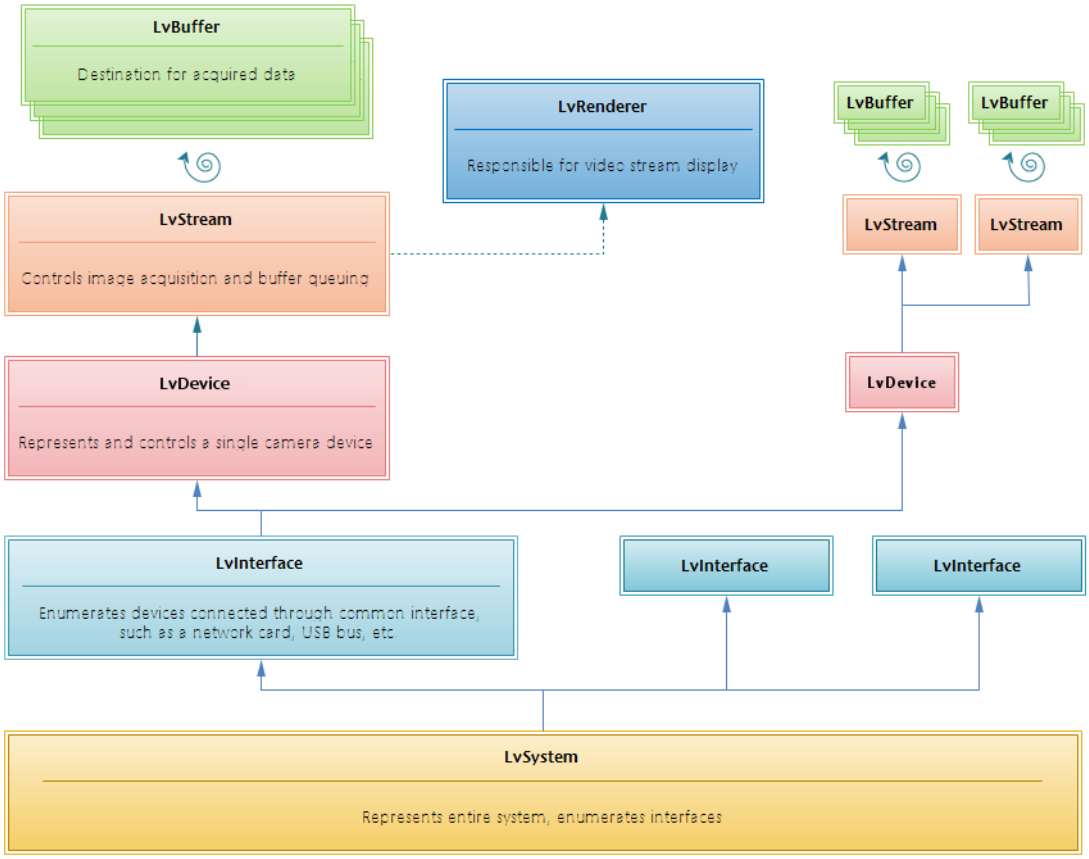


Figure 1.5. Simplon architecture

Notice the hierarchy: The **System** can have one or more *Interfaces*, the **Interface** can provide one or more *Devices*, the **Device** can have one or more *Streams*, the **Stream** holds one or (usually) more *Buffers* and one or more *Renderers*.

1.3.1. The System

This class is the starting point for the usage of the Simplon API in the application. Physically, the System is a **GenTL Provider** library (DLL), which gives an interface to a specific imaging hardware (this library should be delivered by the vendor with its hardware). A GenTL Provider library for Leutron Vision hardware is by default included in the Simplon installation.

Although there could be more systems used concurrently in one application, typical application will have only one system open.

The System provides *Interfaces*.

1.3.2. The Interface

The **Interface** is typically a frame grabber, a network adapter with one subnet or a USB adapter. It is identified by a string ID. The IDs of all Interfaces can be enumerated on the parent System..

The Interface provides *devices*.

1.3.3. The Device

By the **Device** term we refer to a device connected to the Interface, typically represented by a real physical camera. In Simplon, the Device module provides much more functionality, than the GenTL Device.

One Interface can have multiple Devices. The Device is identified by a string ID. The Interface provides the IDs of all Devices found. When you have the Device ID, you can open the Device.

Note that the System, Interface and Device IDs are persistent, so once you know them, you can hardcode them directly in your application. In fact, this is what the *Source code generator* in the Simplon Explorer does. However, this approach has its cons; later on we will discuss other ways how to determine the IDs in alternative ways, so that for example when you replace a defective camera by another one, you do not need to change its ID in your code.

Besides other functionality, the Device module in Simplon provides a *unified image preprocessing* on the default image stream with a chain of processing functions, corresponding to the chain of operations on an RTF version of the grabber or camera. This feature namely serves to the possibility to write universal code for both RTF and non-RTF hardware (on the non-RTF HW the missing processing is done by software). The processing chain typically includes *Bayer Decoding - LUT* (brightness, contrast, gamma, white balance) - *Color Correction - Pixel Format Conversion*.

The Device has one or more *Streams* and the Stream has *Buffers*.

1.3.4. The Stream and Buffer

The Stream module presents a stream of data from the device, typically a stream of images. Usually one device provides only one image stream, but in special cases there can be multiple image streams available, thus the stream is available as a standalone module.

The stream works with buffers in **queues** and **pools**: At the input is a *pool of free buffers*, at the output is a *queue of buffers with acquired images*. The application must pick up buffers from the output queue and after processing return them to the input pool of free buffers. The application cannot supply an arbitrary buffer to the pool. Instead, it must create in advance all the buffers, which will be used. This enables the underlying SW/HW to prepare for the acquisition (upload memory mapping tables etc.).

1.3.5. The Event

The Event module presents an API for events coming from the device or Simplon asynchronously to the application. A typical event is an announcement, that a new image was grabbed and is ready for processing.

The Event module in Simplon actually encapsulates an event queue, giving you a possibility to wait for event and get the event data in one atomic operation, so you do not need to deal with any multithreading protection mechanism in your code — instead your code simply waits for the event and when the wait is done, the event data are already moved to the supplied buffer.

Besides signaling the new image in the stream, the Event module can be also used for other types of events, if they are available:

- an error occurred asynchronously
- events provided by the device (for example messages from a PicSight Smart camera)

Events can be provided by the System, Interface, Device and Stream.

For any event can be created an **internal thread** and registered a **callback function**. In the thread is a loop which waits for the events and calls the callback function for each event and passes to the callback the data got from the event. In case of the event representing the new images, if the callback is not defined by the application, the thread loop automatically returns the acquired image buffers to the input pool (after optional automatic display of the image). Using the internal threads and callback functions enables the code to be operating system independent - your callback routine just needs to take care about being thread safe.

1.3.6. The Renderer

The Renderer module serves for displaying the images in a supplied window. It offers various options (scaled image, tiled display etc.) and is able to convert the image automatically to a displayable format, if it is acquired in a non-displayable format, like 12-bit mono. The implementation of this module is operating system specific — under some circumstances its functionality may be disabled (for example in Linux without XWindows).

1.4. Building Simplon application

Before we get to the Source code generator in Simplon Explorer, let us explain how to create the simplest possible application with Simplon. Understanding this application will help you to get better oriented in the source code produced by the Source code generator.

1.4.1. Opening the library and the system

Let's illustrate the opening of the system on a simple piece of code:

```
LvOpenLibrary();
LvSystem* g_pSystem
LvSystem::Open("", g_pSystem);
if (g_pSystem != NULL)
{
    ...
    LvSystem::Close(g_pSystem);
}
LvCloseLibrary();
```

Before you start using Simplon in your code, you must call the `LvOpenLibrary()` function. And vice versa: before you exit your application, you must call `LvCloseLibrary()` function. These 2 functions serve as fundamental points for initialization and cleanup of the library. They should be called at the points, where the application is normally executed, that means for example in Windows avoid putting them to the process attach and process detach handlers in the `DllMain()`, because at these handlers the code execution has significant limitations.

The code above opens the *default* system and creates a pointer to the `LvSystem` class. We do not need to open multiple systems, so we use a global variable to store the pointer to the system.

Note that for all the Simplon classes is typical, that you cannot use the `new` and `delete` operators directly on these classes (the constructor and destructor are private). Instead, the functions for opening and closing the class instance assure that if the opening is successful, you get a valid pointer, otherwise you get a NULL pointer. Also, the closing functions set the pointer back to NULL. Another advantage is that these functions return a status value, which can clarify the error nature, if the opening or closing fails.

As the first parameter to the `Open()` method, the name of the library containing the GenTL implementation can be specified:

```
LvSystem::Open("lv.gentl.cti", &g_pSystem);
```

If you specify an empty string, the *default* GenTL library name is used — it is specified in the configuration `lv.simplon.ini` file. If this default is not available in the ini file, then the first found library is used. You can also enumerate available systems and give the user a selection - this is what the Simplon Explorer does.

1.4.2. Application basic functions

The simplest application finds the default system, on it finds the first interface, on it finds the first device, opens it and starts acquisition and displaying acquired images in the window. Suppose we have in the application a menu with 4 items:

- Open camera
- Start acquisition
- Stop acquisition
- Close camera

To make the code later extensible for multiple cameras, we establish a simple `CCamera` class representing one device (we call it *Camera* in order not to confuse it with the `LvDevice` class):

```
class CCamera
{
public:
    CCamera();
    ~CCamera();
    bool OpenCamera(HWND hDisplayWnd,
                    LvSystem* pSystem);
    void StartAcquisition();
    void StopAcquisition();
    void CloseCamera();

private:
    LvSystem*    m_pSystem;
    LvInterface* m_pInterface;
    LvDevice*    m_pDevice;
    LvStream*    m_pStream;
    LvRenderer*  m_pRenderer;
    LvEvent*     m_pEvent;
    LvBuffer*    m_Buffers[NUMBER_OF_BUFFERS];
    HWND         m_hDisplayWnd;
};
```

And here is the implementation of the corresponding 4 functions, one for each menu item. As already mentioned, the application opens the system at startup and stores a pointer to it in `g_pSystem`, so we have it available. Note that for simplicity *the error handling is completely omitted*. In a real application this would of course not be a recommended approach.

```
01  CCamera::CCamera()
02  {
03      m_pSystem     = NULL;
04      m_pInterface  = NULL;
05      m_pDevice     = NULL;
06      m_pStream     = NULL;
07      m_pRenderer   = NULL;
08      m_pEvent      = NULL;
09      memset(m_Buffers, 0, sizeof(m_Buffers));
10      m_hDisplayWnd = NULL;
11  }

13  CCamera::~CCamera()
14  {
15      if (m_pDevice != NULL) CloseCamera();
16  }

18  bool CCamera::OpenCamera(HWND hDisplayWnd,
19                           LvSystem* pSystem)
20  {
21      if (m_pDevice != NULL) CloseCamera();
22      m_pSystem = pSystem;
23      m_hDisplayWnd = hDisplayWnd;
24      std::string sDeviceId;
25      m_pSystem->OpenInterface("", m_pInterface);
26      m_pInterface->UpdateDeviceList();
27      m_pInterface->GetDeviceId(0, sDeviceId);
28      m_pInterface->OpenDevice(sDeviceId.c_str(),
29                               m_pDevice, LvDeviceAccess_Exclusive);
30      return true;
31  }

33  void CCamera::StartAcquisition()
34  {
35      if (m_pDevice == NULL) return;
```

```
36      m_pDevice->OpenStream("", m_pStream);
37      m_pStream->OpenEvent(LvEventType_NewBuffer, m_pEvent);
38      for (int i=0; i<NUMBER_OF_BUFFERS; i++)
39      {
40          m_pStream->OpenBuffer(NULL, 0, NULL, 0, m_Buffers[i]);
41          m_Buffers[i]->Queue();
42      }
43      m_pStream->OpenRenderer(m_pRenderer);
44      m_pRenderer->SetWindow(m_hDisplayWnd);
45      m_pRenderer->SetAutoDisplay(1);
46      m_pEvent->StartThread();
47      m_pDevice->AcquisitionStart();
48  }

50  void CCamera::StopAcquisition()
51  {
52      if (m_pStream == NULL) return;
53      m_pDevice->AcquisitionStop();
54      m_pEvent->StopThread();
55      m_pStream->CloseEvent(m_pEvent);
56      m_pStream->CloseRenderer(m_pRenderer);
57      for (int i=0; i<NUMBER_OF_BUFFERS; i++)
58          if (m_Buffers[i] != NULL)
59              m_pStream->CloseBuffer(m_Buffers[i]);
60      m_pDevice->CloseStream(m_pStream);
61  }

63  void CCamera::CloseCamera()
64  {
65      if (m_pDevice == NULL) return;
66      StopAcquisition();
67      m_pInterface->CloseDevice(m_pDevice);
68      m_pSystem->CloseInterface(m_pInterface);
69  }
```

Line 01-11: The class constructor sets the initial pointer values to `NULL`. It is important to know, that all `Close` methods of Simplon classes set the value of the referencing pointer back to `NULL`. So for example at line 67 the `m_pDevice` is set to `NULL` in the `CloseCamera()` call (that is why the parameter is passed as reference to these methods).

Line 18-23: The handle of the window used for display of the images and the pointer to the already open system are passed as parameters. These are stored for later usage.

Line 25: Open the first available Interface — Simplon enables to use an empty string as the Interface ID, meaning *use the first found* — this approach has almost no real usage — whenever multiple interfaces would appear, you would not be sure which one will be opened; but for the first test application it may be sufficient (if not, the Source code generator later on will show you which Interface ID to use).

Line 26: Before we can ask for the list of devices available, we must first call the `UpdateDeviceList()` method. This method creates an internal list of Devices, which can be retrieved by the `GetDeviceId()` function. As a parameter is required the pointer to Interface.

Line 27: Once we have the Device list available, we can ask for the Device IDs in the list. Here, for simplicity, we simply use the first found Device, that means the Device at index 0 (again, this approach is suitable only for test purposes and would not fit for cases when multiple devices are available). After this line we have the Device ID in the `sDeviceId` string.

Line 28–30: The Device is open for the exclusive access and the pointer to the `LvDevice` class instance is returned in the `m_pDevice`. Now the Device features could be read and modified - for now we will leave the Device features untouched (how to modify the features will be shown in next chapters). For simplicity we do not care about the error status and return `true`.

Line 33: The `StartAcqusition()` method creates an environment for receiving the images and then sends to the device a command to start acquiring and sending the images.

Line 36: For the image acquisition we need to create a Stream. Normally, the Stream ID would be needed — if we leave the Stream ID as empty string, the default stream type is used, which is always the *default image stream*.

Line 37: We also need to create an Event - it enables the application to be notified about new grabbed images in the output queue (now we will not explicitly use this notification, but we will show it in the next chapter) and to create an internal thread for image handling.

Line 38-42: For holding the images, at least one Buffer must be created. Usually more Buffers are created (in our case 10), so that one Buffer can be filled by a new image while another Buffer is being processed. The first parameter = `NULL` in the `OpenBuffer()` method says that the image buffer should be allocated by Simplon (you can also pass a pointer to your own buffer here) and the second parameter = `0` says that image buffer size should be automatically determined by Simplon from the Device parameters.

Line 41: The `Queue()` method puts the allocated Buffer to the acquisition *input pool*. In this pool are free Buffers, which are taken by the Device, filled with the image data and placed to the *output queue*, from which the application can pick them up, process, and then pass them to the *input pool* again. In this first version of the application we will leave this process on automated internal Simplon mechanism (no callback defined).

Line 43-44: For displaying the images in a window, we create a Renderer. This class is able to do automated image display. For now, we leave the renderer settings on default. The line 44 is operating system dependent — for Windows you pass a window handle, for Linux with XWindows you would pass a pointer to *display* and a *window* handle.

Line 45: Enables the automatic image display when the image is acquired, so we do not need to create any callback for image handling for now.

Line 46: The `LvEvent` class enables to start a thread, which processes the waiting on the event. The start (and stop) has to be done explicitly, because you may also need to use your own thread, instead of this one, so any automatic thread starting would be undesirable. Usually using this thread is connected with a callback function, which the thread calls for each image; but for now the callback function is not used. In such case (when no callback function is specified) it is internally assured that the rendered Buffer is automatically passed to the *input pool* of free buffers after optionally displaying it, so we do not need to take care about it. However, in the next chapter, we will establish own callback function, and then it will be us, who will become responsible for returning the Buffers back to the input pool.

Line 47: Now everything is prepared for acquisition, so we can start it, by this method.

Line 53–60: Acquisition is first stopped and then the Event, Renderer, Buffers and Stream are closed, in the reverse order than in the previous method.

Line 66–68: Closes and frees the Device instance.

1.4.3. Error checking

Note again, that the code shown in this document is mostly *without any error checking*. In a real code, you should always check the return values of called Simplon methods, at least at places, where you can expect some failure. For example, if there is no device available, the function at line 27 returns an error value, and it does not make sense to continue with the next code.

Most Simplon functions and methods return a *status value*, indicating the success or error status. The success is indicated by returning the value `LVSTATUS_OK` value (which is 0). In the Simplon C++ Class Library you can also switch on throwing exceptions on errors. In the Simplon .Net Class Library the exceptions are switched on by default. The error status should always be checked in a real application.

1.4.4. Adding a callback function

Your application will usually need not only to display the acquired images, but also to do something with them.

Image processing is usually done in a separate thread, so that the main application thread can still respond to the user control. You can create your own thread and use the `LvEvent::WaitAndGetData()` method for getting the new images from the output queue.

Or you can let Simplon to create such thread internally for you and specify a **callback function**, which is called for each buffer picked from the output queue. We will show this possibility in this chapter. The callback function for the *new buffer event* (= new image was acquired) has a defined shape in the `LvEventCallbackNewBufFunct` typedef:

```
typedef void (LV_STDC* LvEventCallbackNewBufFunct) (LvHBuffer hBuffer,
                                                    void* pUserPointer,
                                                    void* pUserParam);
```

The callback function cannot be directly a method of a class, so we will utilize the `pUserParam` to pass the pointer to a `CCamera` class instance:

```
void LV_STDC CallbackNewBufferFunction(LvHBuffer hBuffer,
                                       void* pUserPointer,
                                       void* pUserParam)
{
    CCamera* pCamera = (CCamera*) pUserParam;
    pCamera->CallbackNewBuffer((LvBuffer*) pUserPointer);
}
```

As you can see, the callback function simply calls the `CallbackNewBuffer()` function of the `CCamera` class - in this way we can utilize single global callback function for multiple class instances.

And here is the `CallbackNewBuffer()` implementation:

```
void CCamera::CallbackNewBuffer(LvBuffer* pBuffer)
{
    m_pRenderer->DisplayImage(pBuffer);
    pBuffer->Queue();
}
```

The callback handler above does nothing else than what was before done automatically, i.e. it displays the image and puts the Buffer back to the *input buffer pool* (by the `LvBuffer::Queue()` method). It is important to know that once we start using the callback function, we have the responsibility to return the processed Buffer to the input buffer pool (because when the callback function is defined, this is no longer done internally). It is also important to keep on mind that this method is called from a different thread, so it is executed asynchronously to your main application thread.

To make the callback functional, the `StartAcquisition()` method had to be slightly modified: On line 45 the original line for setting the *AutoDisplay* must be removed and the following line of code added:

```
m_pEvent->SetThreadCallbackNewBuf(CallbackNewBufferFunction, this);
```

The second parameter is passed as a `pUserParam` in the callback function, so as already mentioned, the `CCamera` class instance can be easily determined.

Of course, we do not create the callback only to do the same, as could done by default internally in Simplon — in a real case we will need to have an access to the image data, to be able to process them. Here are 2 lines to be added at the beginning of the `CCamera::CallbackNewBuffer()` in order to get a pointer to image data

```
    void* pData = NULL;
    pBuffer->GetPtr (LvBuffer_Base, &pData);
```

For image processing, we will also need to get some more information, for example image width, height, pixel format etc. These items are device features and we will discuss how to get and set features in the next chapter.

1.5. Working with features

The device parameters and controls are commonly referred as *features* in the GenICam terminology. The **features** are essential building stones of the applications using Simplon API.

You might wander from where the Simplon Explorer knows, which features are available for the connected device. The answer is simple: thanks to the *GenICam* standard, the device is able to report all its features to the user application. The user application is thus becoming independent on the particular device: Simplon is designed to be fully compatible with existing standards and thus it is possible to write applications compatible with a wide range of devices (even of various vendors), of course with a condition, that these devices are also supporting the necessary standards.

The features can be read and/or written, and a special type of feature - the *command* - can be executed. Features can be displayed in a hierarchical tree, like you can see in the Simplon Explorer application.

A lot of useful detailed information about features in general is in the Understanding the feature tree chapter in the Simplon getting started manual. Descriptions of particular features of given camera are in the Camera features and their control in the PicSight-GigE user manual. Before you continue with reading this manual, we strongly recommend you to read these chapters.

This chapter gives just an introduction to features in Simplon API, further description is in the Section 2.4, “Features” [p. 50] chapter.

1.5.1. Feature name and display name

For programming, you will need to know the *feature name*. The name you see in the feature tree is not the feature name — there you see the *display name*, which is usually more verbatim. The following figure shows the difference: While the **display name** is *X Offset*, the **name** is *OffsetX* (without spaces). For your convenience, the *name* is displayed in the info panel and in the next chapters you will see that by this name you can obtain a handle to the feature (so called generic access). Simplon also offers symbolic constants for features, which are derived from their names, for the example above it would be the `LvDevice_OffsetX`.

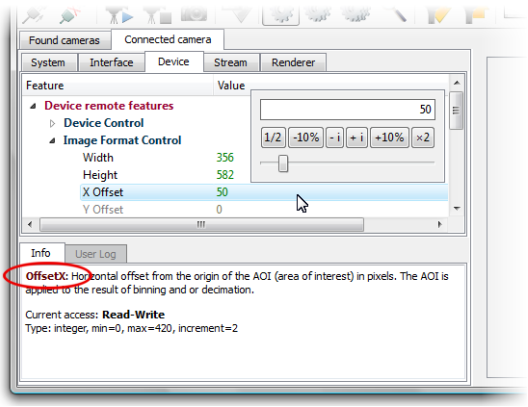


Figure 1.6. Simplon Explorer: Feature name

1.5.2. Feature naming conventions

A small set of features are *mandatory* (must be always implemented), like the *Width*, *Height* and *Pixel Format*. The naming of the features is specified in the SFNC —*Standard Feature Name Conven-*

tion, which unifies the names of the features on the cameras from different vendors. For example you can be sure that each device will have the *PixelFormat* feature named in the same way.

However, most of the features are not mandatory, they are optional and need not be implemented. For most of these optional features exists also a recommended naming convention, so if the feature is implemented, it is probable, that it has the same name as the same feature by another device from a different vendor. For the Leutron Vision grabbers and cameras we put stress on using standard naming conventions wherever possible.

Finally, the camera can provide also unique features, which do not yet exist in SFNC. On our hardware, the names of the unique features are prefixed with `Lv` (for example: `LvTriggerCaching` or `LvBayerDecodingAlgorithm`).

As only a small set of features is really mandatory, the application, which is designed to work with arbitrary cameras, must always check the availability of particular feature with which it wants to work.

1.5.3. Generic access to a feature

The **generic** (= universal) way how to obtain a possibility to work with a feature is to obtain an ID of the feature *from its name*. The following sample code shows how to get the `SensorWidth` feature value:

```
int32_t iSensorWidth;
LvFeature iFeatureId;
if (m_pDevice->GetFeatureByName (LvFtrGroup_DeviceRemote,
                                 "SensorWidth",
                                 &iFeatureId) == LVSTATUS_OK)
{
    m_pDevice->GetInt32(iFeatureId, &iSensorWidth);
}
```

A disadvantage of the generic access is that for accessing a single feature you need to insert a quite complex code and that a typing error in the ID name will not be discovered by the compiler, but only in the run-time (if at all).

1.5.4. Access through a predefined constant

To make the access more comfortable, Simplon provides also an alternative way, how to work with the features: Simplon provides a predefined ID value for all features supported by Leutron Vision hardware (that means all GenICam mandatory features + all GenICam optional features supported on Leutron devices + all Leutron custom features) in the `LvFeature` enumerations (see the `lv.simplon.enum.h` file). With this predefined value you do not need to ask for the ID - you simply use the predefined one:

```
int32_t iSensorWidth;
m_pDevice->GetInt32(LvDevice_SensorWidth, &iSensorWidth);
```

Another advantage is that the same constants (if applicable) are used in the API of the PicSight Smart cameras.

1.6. Simplon source code generator

If you have read the previous chapters, you should be now familiar with the basic Simplon concepts and there should be no problem to understand the usage of the Source code generator, available in Simplon Explorer.

Simplon Explorer offers 3 ways how to help you with the creation of the source code:

- Showing in the Info panel the **source code for each selected feature** in the tree. This helps you in case you need to add usage of some feature to an already functional application - instead

of searching for the feature in the manual, you can directly copy and paste the needed code to your application.

- **Showing a code snippet** for the selected branch in the device feature tree or for the whole device feature tree. This extends the previous option by providing a snippet of code for multiple features, ready to be copied and pasted to your application.
- **Creating the whole project**, which contains the current device configuration and is immediately compilable. This is a good starting point for exploring how to create applications using Simplon.

Let's have a look at these possibilities in detail.

1.6.1. Source code in info panel

To see the source code in the panel, you must switch this option **on**:

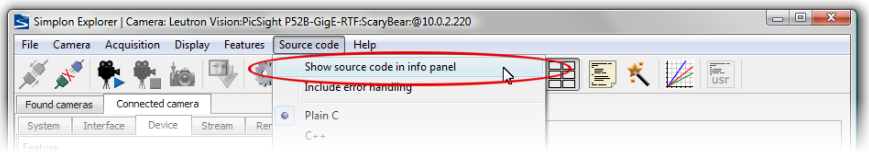


Figure 1.7. Simplon Explorer — Switching on Source code display

You can also see in the menu that you can select the **language** of the Simplon API. Also, as an option, you can select, whether the code should contain **error handling** or not.

When a feature is selected in the tree, the corresponding source code is displayed in the info panel:

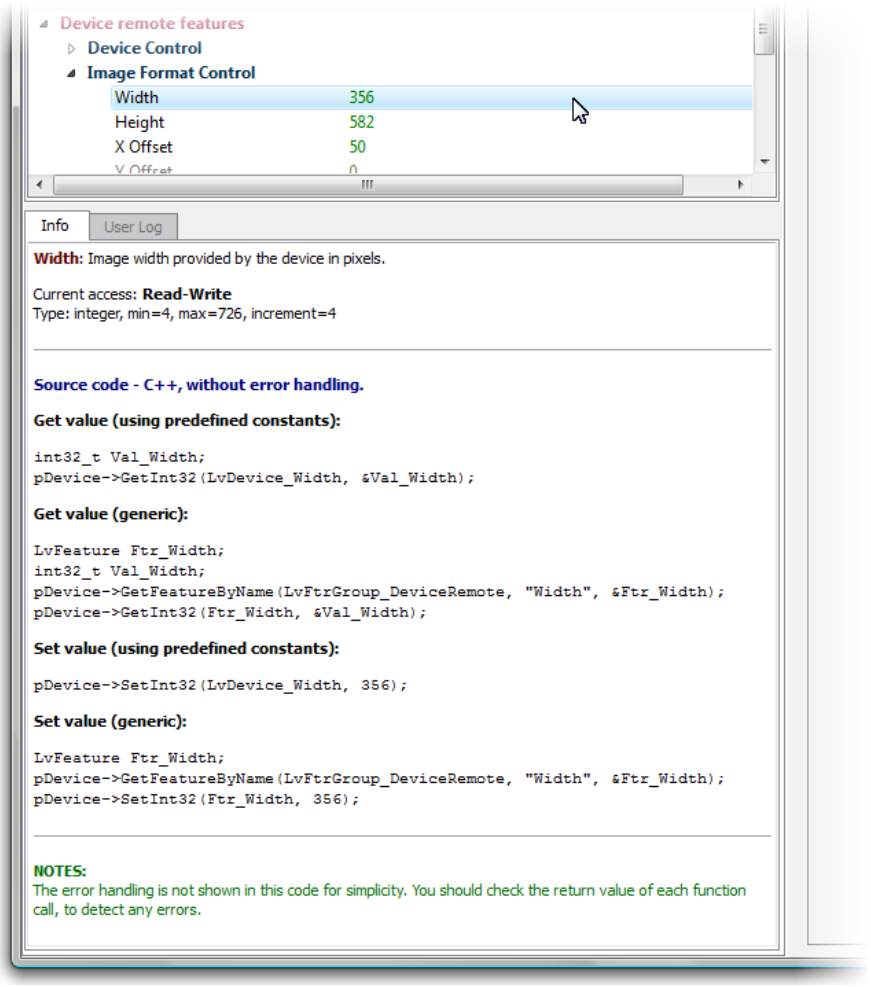


Figure 1.8. Simplon Explorer — Showing source code for a feature (Width)

In the image above, the source code is displayed for the *Width* feature. This feature can be accessed using either the *generic* approach (via a string identifier), or in a simpler form, using a Simplon *predefined constant*. The code is shown for both getting and setting the value - even if the value is currently read-only. In this way you get immediate code lines for the selected feature.

Do not forget, that some features may depend on a parent selector — in your code the selector must be set as well. If the feature is dependent on a selector, it is mentioned in the info panel ("*This feature is selected by...*"). In such case your code should first set the proper selector and then the feature(s) under the selector.

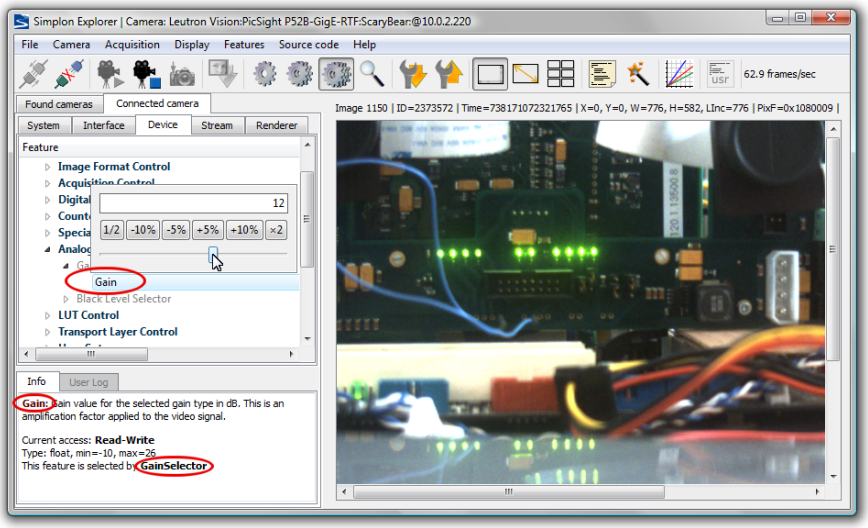


Figure 1.9. Simplon Explorer — Setting gain — note about the Gain selector

The *code snippet* (see the next chapter) includes also the selector setting (if you select the appropriate branch).

1.6.2. Creating a code snippet

You can open a dialog box with a code snippet with the following item in the menu:

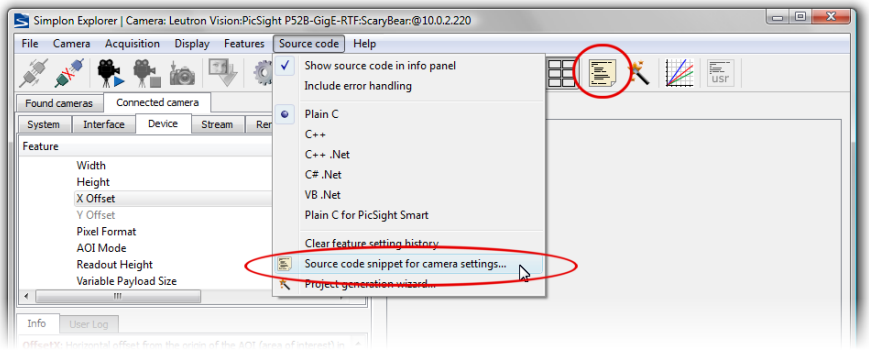


Figure 1.10. Simplon Explorer — Menu for code snippet

However, before you open the code snippet dialog, you should consider the following issues:

Display level

The code snippet is made from **displayed features**, that means, the not displayed features are *not included* in the code snippet. Thus, you might want to change the **Display Level** from *Beginner* to *Expert* or even *Guru*, to be sure the snippet shows all the desired features:

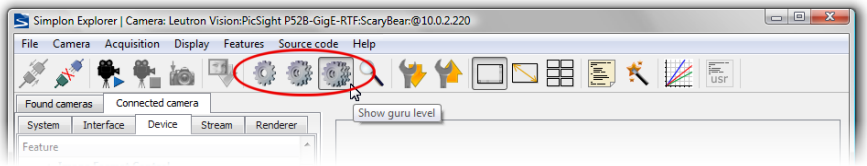


Figure 1.11. Simplon Explorer — Toolbar button for display level

History

Whenever you set some feature in the tree, it is automatically recorded to a **history list**. Then the code snippet can be generated from the history of settings. This approach has an advantage, that it includes only actions needed to achieve some device settings. Another advantage is that is also includes commands (`LvDevice::CmdExecute()`), which are normally not included. An example of a command can be the loading of a user set configuration to the device. The history is recorded from the point, when the device was open.

If you want to explicitly clear the history list and start over, you can do so in the menu:

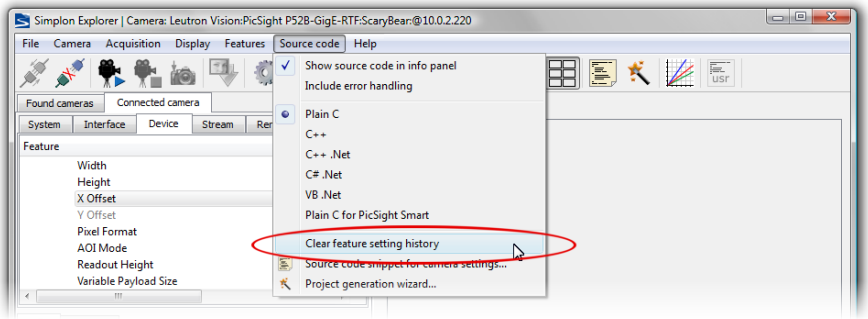


Figure 1.12. Simplon Explorer — Menu for clearing the history

When you open the **Code Snippet dialog**, it looks as follows:

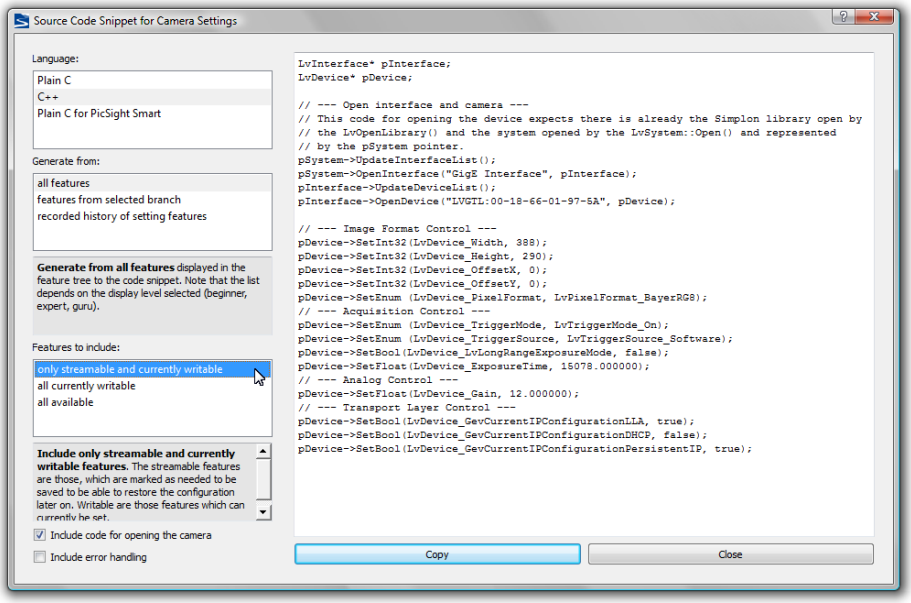


Figure 1.13. Simplon Explorer — Code snippet dialog

You can select again the language and error handling, like in the case of code in the Info panel. Additionally, you can switch on the *Include code for opening the device* option, which inserts at the beginning the code for opening proper Interface and Device.

The **Generate From** list box includes up to 3 options:

1.6.2.1. Generate from: All features

The whole device feature tree is used. Recall that this option also depends on the display level - the features not displayed are also *not included* in the code snippet.

A general question is which features should be included in the code snippet - this can be selected in the **Features to include** list box:

- **Only streamable and currently writable**

  Normally, we can assume that the application wants to set those features, which are necessary to be set in order to configure the device to a desired mode. Such features will for example include image *Width*, *Height* and *Pixel Format*, but should not include for example the *Device User ID*. The GenICam standard provides for each feature an attribute called "**streamable**" - this attribute says the feature needs to be saved in order to save the device configuration. So the *streamable* attribute may be a good guide for selection, which features are to be set in the code snippet as well.

  Usually, it also does not make sense to set the features, which are not writable, so only writable features are included.

  Note that the *streamable* attribute may be missing on the features of devices of other vendors, in such case you will get an empty source code snippet.

- **All currently writable**

  In some cases you might want to see in the code also features which do not have the *streamable* attribute - then you can use this second option. Be careful with this option: it is then on you to manually remove from the code snippet the features, which does not make sense to set.

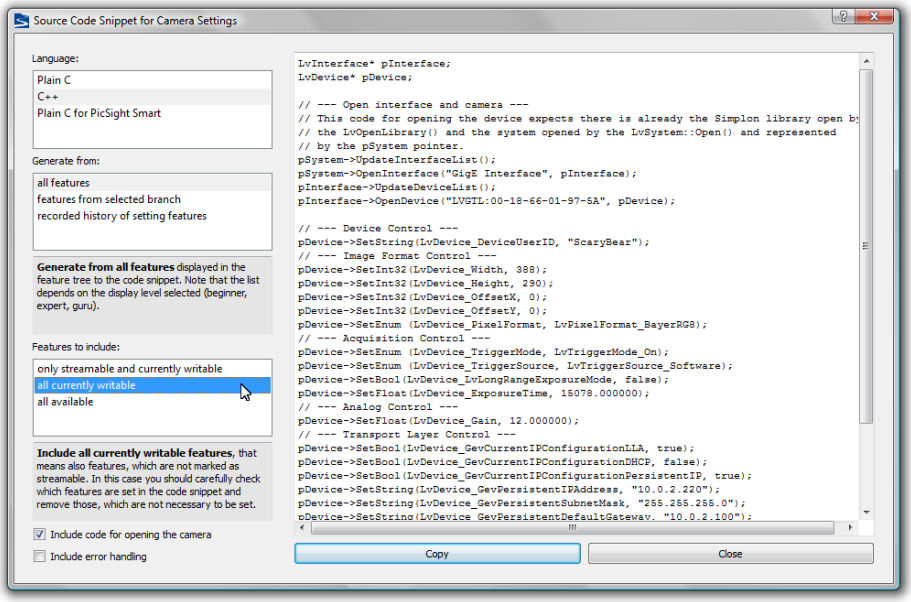


Figure 1.14. Simplon Explorer — Code snippet dialog (All writable)

- **All available**

  You might even want to see the code for all available features (which are not permanently read-only), even if it does not make sense to use such code snippet directly, as it surely would produce a lot of error states. Again, it is on you to manually remove all the features, which do not make sense to set.

1.6.2.2. Generate from: Features from selected branch

Suppose that you want to get only a code for the I/O settings. Then you can select the *Digital I/O Controls* in the tree (before opening the Code Snippet Dialog) and then in the dialog select the

option *Features from selected branch* (note that this option is hidden if no branch is selected). Only items under this branch are used for code generation.

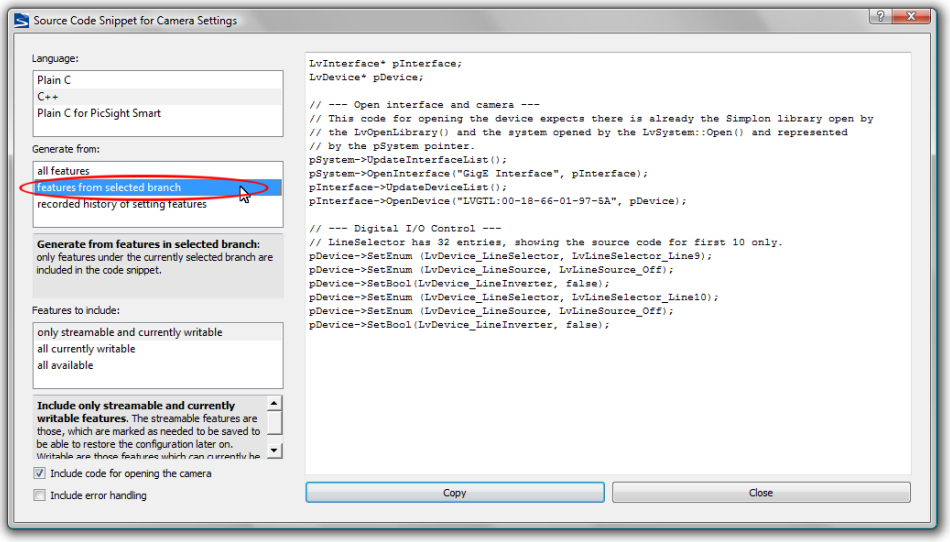


Figure 1.15. Simplon Explorer — Code snippet dialog (features from a branch)

1.6.2.3. Generate from: Recorded history of setting features

If you set at least one feature in the feature tree (so the history is not empty), the option *Recorded history of setting features* becomes available - the snippet then contains only the features, which have been set (including *Commands*, like *User Set Load*. The *Features to include* options are not available in this mode.

1.6.3. The Project Generation Wizard

The wizard dialog box can be opened in menu:

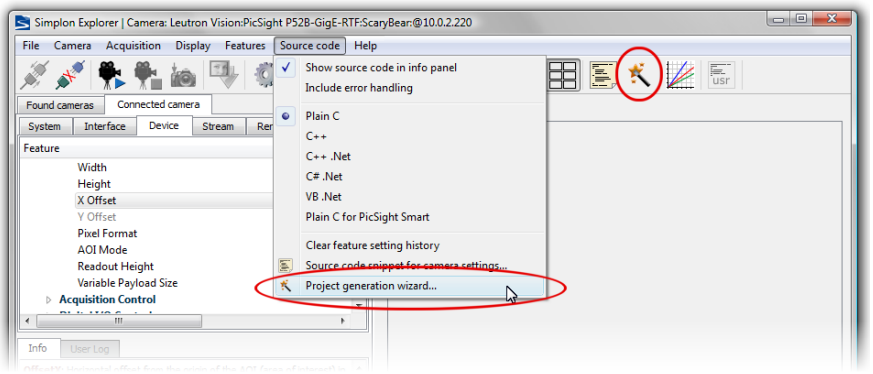


Figure 1.16. Simplon Explorer — Menu for the source code wizard

The first thing you should select in the wizard dialog box, is the language. After the language is selected, Simplon finds all available project **templates** and displays a list of them (on our screenshots only 2 templates are available).

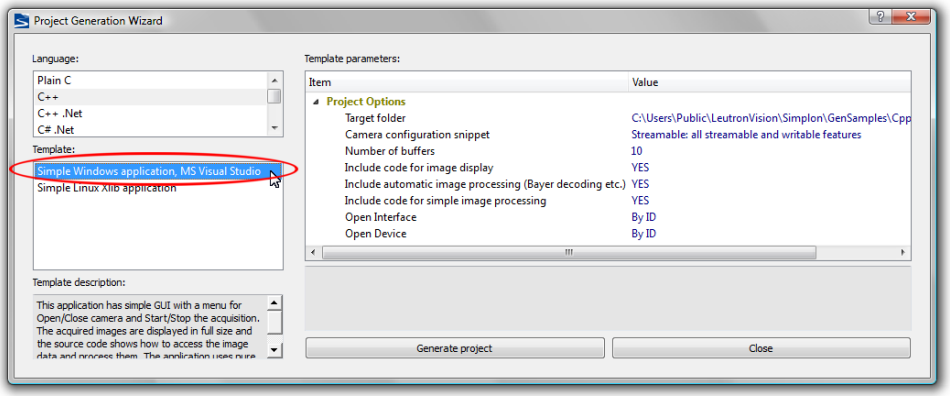


Figure 1.17. Simplon Explorer — Select the template in the wizard

The template parameters are displayed at the right part. There is possible to select:

- The **target folder** (where the generated project will be stored):

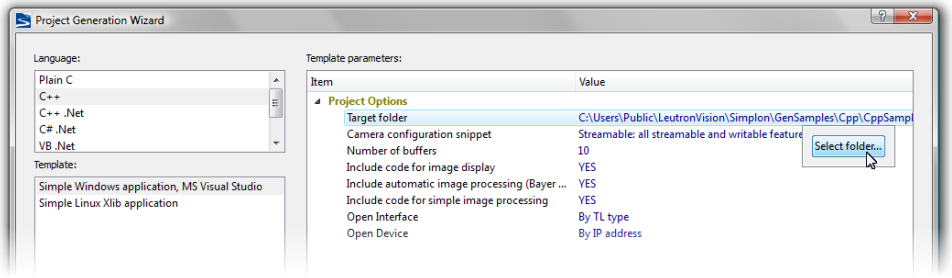


Figure 1.18. Simplon Explorer — Select the target folder in the wizard

- The **type of device configuration snippet**

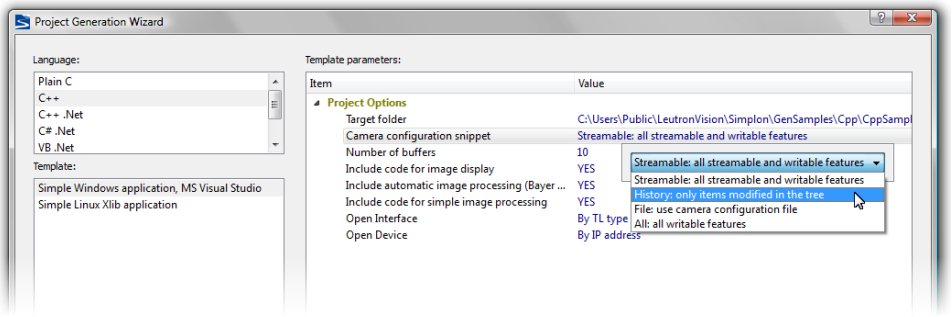


Figure 1.19. Simplon Explorer — Configuration of the wizard

The first 2 options you see in the combo box are already known from the *Code Snippet* chapter. The third option - *Use device configuration file* uses the device *Save/Load Settings* mechanism instead of direct settings the features. The current device settings are saved to a file and in the code snippet is a call to `LvDevice::LoadSettings()` function, loading the configuration to the device.

- **Other settings** — these settings are provided by the template itself, so the might differ from template to template:

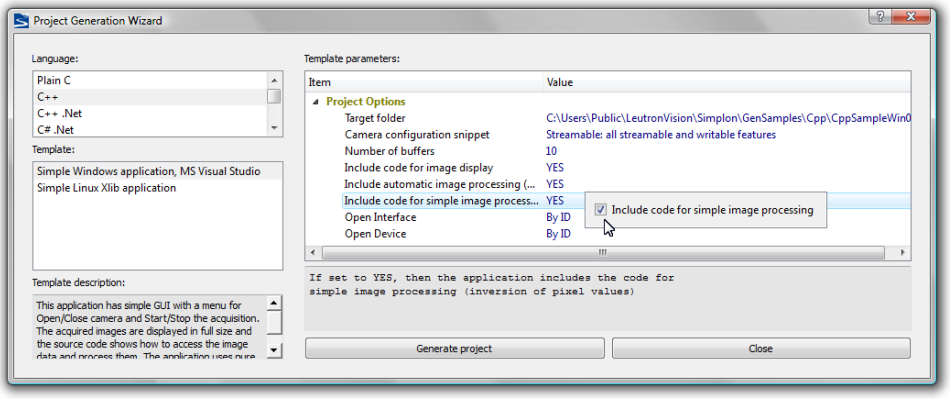


Figure 1.20. Simplon Explorer — Template options in the wizard

When you press the *Generate project* button, the project is generated and you get the following notification:

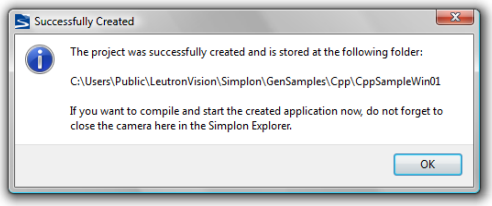


Figure 1.21. Simplon Explorer — Confirmation message of the wizard

If you want to compile and **run** the created application, it is **important to disconnect the camera in the Simplon Explorer**, because it is open for exclusive use and any attempt to open it from another application would fail.

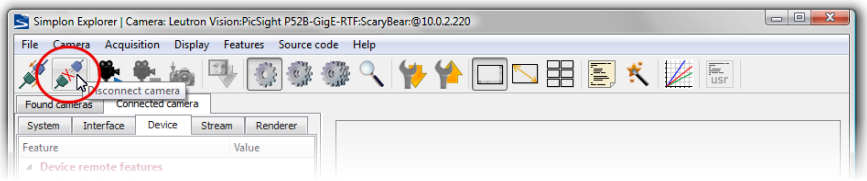


Figure 1.22. Simplon Explorer — Disconnect the camera

1.6.4. The usage of generated code

Keep in mind, that the Source Code Generator is just a support tool. When you generate a project, you should understand its contents in order to check, how it fits to your needs and what is needed to be changed. So the project is rather a startup than the final one — you may need to insert functionality not provided by the generator, for example some code for changing features during the acquisition (like *Exposure* or *Gain*). In such case you can utilize the Code Snippet to create additional code fragments and copy and paste them in the application.

Note also that by default the generated code places to the code directly the device ID (you can change this in the template options). This might not fit to a real application, because every piece of device has its unique ID, which means when you replace a defective device by another one, you would have to change and recompile the source code, even is the device is of the same type. So instead of direct opening the device by ID may be more suitable to search for the device by

the IP address or by User ID. This topic is discussed in Section 2.5, “Writing maintainable applications” [p. 57].

2. Programming with Simplon

The previous chapters were giving you a quick introduction to the work with the Simplon API. In the following chapters you will get a deeper and more exact insight into the Simplon API architecture and usage.

2.1. Simplon files and environment

This chapter contains information how to setup your compiler in order to work successfully with Simplon.

2.1.1. Windows

The Simplon Setup creates an environment variable `LV_SIMPLON`, which points to the installation root folder. This variable is recommended to be used instead of absolute path, for example in MS Visual Studio project setting you can utilize the `$(LV_SIMPLON)` literal to substitute current Simplon root folder.

Several folders are created under the installation folder:

- `BIN` — this folder contains all executables and compiled libraries. The Setup sets the PATH to this folder, so that your application need not reside in the same folder. On Windows till the XP version the BIN folder is also used for some files, which might be changed — typically the `lv.simplon.ini` file. In Windows Vista and later these files are located in another folder, which is not write protected — see Simplon getting started.

- `INCLUDE` — in this folder are header files for your compiler. Typically, in MS Visual Studio you will need to add `$(LV_SIMPLON)\Include` to the *Project Properties — C/C++ — Additional Include Directories*.

- `LIB` — in this folder are import libraries. For Simplon you will need to link with the `lv.simplon.lib` file. You may also need to link with `lv.simplon.ini.lib` in case you want to utilize the OS independent INI file functionality

  and with `lv.simplon.imgproc.lib`, in case you want to utilize the standalone image processing library.

  Again, the `$(LV_SIMPLON)\Lib` is recommended to be used instead of an absolute path in MS Visual Studio in the *Project Properties — Linker — Additional Library Directories*.

2.1.1.1. The stdint.h file

The Simplon uses the `int32_t`, `uint32_t`, `int64_t`, ... types from the `stdint.h` file, which is almost standard among compilers. Unfortunately, not among all, the exceptions include also the MS Visual Studio.

Simplon till version 1.00.017 installed this file as renamed to `stdint_.h`, so you could simply rename this file back to `stdint.h` in case your compiler did not have it available.

As this solution was a source of confusion, since the version 1.00.018 Simplon installs directly `stdint.h` (not renamed) to the `Include` subfolder. In case this file would clash with your version of the file (which is anyway not likely), please remove the `stdint.h` file from the Simplon `Include` folder.

More info at: http://en.wikipedia.org/wiki/Stdint.h and Google search for: *visual studio stdint.h*.

2.1.1.2. Setup of Simplon .Net Class Library

The following note applies to Windows XP and earlier, not to Windows Vista, Windows 7 and later.

Simplon needs to have 2 environment variables set. Normally these variables are set dynamically in Simplon with a DLL delay load mechanism, so that it is assured no clash can happen with any 3rd party packages, using the same variables. However, the .Net Framework in Windows XP ignores these variables, so they must be set manually (Windows Vista and newer already do not have this problem):

The `GENICAM_ROOT_V2_1` should point to the SIMPLON_ROOT`\bin\GenICam`, where SIMPLON_ROOT is the folder, to which Simplon was installed. Typically:

```
GENICAM_ROOT_V2_1=C:\Program Files\LeutronVision\Simplon\bin\GenICam
```

The `GENICAM_CACHE` should point to the SIMPLON_DATA`\Cache` (which on Windows XP is the same as SIMPLON_ROOT`\Bin`) Typically:

```
GENICAM_CACHE=C:\Program Files\LeutronVision\Simplon\bin\Cache
```

The safest way is to run Simplon Explorer and then have a look at the `lv.simplon.log` file, the correct paths are visible there (search for the "Setting env" string).

Note that if you use on the same PC some 3rd party software, which also usese the Gen-i-cam libraries, the above stated variables may already exists and point to a different location. If the libraries at this location are of different or custom version, unexpected side effects can appear.

2.1.2. Linux

The default installation location of the Simplon package under Linux is `/opt/simplon`. The files most important for application development are located in following subdirectories:

- `include` — contains the Simplon header files.
- `lib` (`lib64`) — contains the Simplon API libraries (shared objects). All Simplon API applications link with `liblv.simplon.so`, applications using Simplon Image Processing Library link additionally with `liblv.imgproc.so`. The other shared objects are their dependencies and are not intended for direct linking to applications.
- `bin` — contains executables, such as Simplon Explorer, Simplon Settings or Simplon GigE Vision Configuration Tool.
- `cti` (`cti64`) — contains the Simplon GenTL Producer, important for developers accessing the cameras through GenTL interface rather than through Simplon API.

A typical command building a Simplon API based application under Linux (32-bit), could look like

```
g++ my_app.cpp -I/opt/simplon/include -L/opt/simplon/lib -llv.simplon
```

The Simplon setup adds the Simplon API library links to the dynamic linker configuration, so the libraries will be properly located in the system during runtime.

The dynamic linker configuration is performed through the `/etc/ld.so.conf.d/` mechanism, which works well on all major Linux distributions. When using a Linux system not using the `/etc/ld.so.conf.d/` configuration, you will have to configure the dynamic linker “manually”. Alternatively, you could adjust the runpath/rpath for your application when building it, so that it can locate the Simplon API libraries, for example:

```
g++ my_app.cpp -I/opt/simplon/include -L/opt/simplon/lib -llv.simplon \
    -Wl,-rpath,/opt/simplon/lib
```

Required build/runtime environment: compiler g++/gcc 4.1 and newer, libstdc++ v. 6 (GLIBCXX 3.4.5), pthreads implementation NPTL.

2.2. Types of Simplon APIs

To support a big variety of compilers and environments, Simplon comes with 3 types of the API:

- **Plain C** — this is the base API, the other APIs are actually only wrappers utilizing internally this base API. On this API is kept the backward compatibility — all the existing functions and constants are preserved, so that once an application is compiled, it should work also with newer versions of Simplon, than is the one, with which the application was compiled. Thus it is possible to update the Simplon libraries without need to recompile your application.
- **C++** — this is a wrapper class around the base API. The wrapper class enables more comfortable programming (is more suitable for tools like Intellisense in MS Visual Studio, enables inheritance).
- **.Net Framework Class Library** — this is a wrapper for compilers in MS .Net Framework — namely for C++, C# and Visual Basic.

2.2.1. The Plain C API

All the functions have the standard form (no classes), so the library with the functions is usable not only from compilers, but also from many other tools (third party image processing SW is often capable of using such library). The application should include the `lv.simplon.h` file in the source code.

2.2.2. The C++ Class Library

The C++ API is implemented as a simple set of wrapper classes, the source code of which is supplied in the `Include` directory in the `lv.simplon.class.h` and `lv.simplon.class.cpp` files. Include the `lv.simplon.class.h` in your source code and add the `lv.simplon.class.cpp` to your project. If you modify these files, you can easily move and keep them with your project, because the backward compatibility is kept on the Plain C API, so you can recompile your application (including the modified `lv.simplon.class.cpp/h` files) with a new version of Simplon.

By default, the class library uses the `std::string` type for all functions, which **get** a string value. This makes the work more comfortable.

```
LvStatus LvModule::GetString (LvFeature Feature, std::string& sValue);
```

However, if you prefer not to use the `std:string` type, all the functions are also available in the "classic" form, that means with a `char*` pointer to a buffer and the size:

```
LvStatus LvModule::GetString (LvFeature Feature, char* pszValue, size_t iSize);
```

You can avoid the inclusion of the standard library by removing the `#define LV_USE_STDLIB` at the beginning of the `lv.simplon.class.h` file.

Note that for **setting** a string value only the "classic" form is available, that means when using a std::string, it must be passed with the `.c_str()` pointer. The reason for this is that the compilers do not recognize the difference between `std::string` and `const char*` as valid for function overloading.

```
LvStatus LvModule::SetString (LvFeature Feature, const char* pszValue);
```

2.2.3. The class library for MS .Net Framework

The Simplon .Net Class Library wraps all Simplon API classes to their managed counterparts. The library uses in its interface mostly the .Net Framework managed types (`Int32`, `UInt32`, `Int64`,

`Boolean`, `Double`, `Array`), only in case the conversion is not possible (would be CPU power or resources consuming), the pointer to unmanaged data (as `IntPtr`) is obtained. This applies namely to the image buffers and the Source Code Wizard samples show how to access such unmanaged buffers.

Also the *Image Processing Library* is included in this .Net Class Library. All-in-one, you only need to use one DLL in your projects.

The library has all its API in the **LeutronVision.Simplon** namespace.

The classes are named in the same way as in the C++ Class Library: `LvLibrary`, `LvSystem`, `LvInterface`, `LvDevice`, `LvStream`, `LvBuffer`, `LvEvent`, `LvRenderer`.

The `LvLibrary` contains only static methods, so that they can be called without need to create an instance of the class.

The Image Processing Library offers the following classes: `LvipImage`, `LvipLut`, `LvipColorCorrectionMatrix`, `LvipConvolutionMatrix`, `LvipOverlay`

2.2.3.1. The .Net Class Library files

The class library is stored in a single DLL file, which exists in 4 versions:

- `LeutronVision.Simplon.dll` — 32–bit version for .Net Framework 2.0, 3.0 and 3.5
- `LeutronVision.Simplon.x64.dll` — 64–bit version for .Net Framework 2.0, 3.0 and 3.5
- `LeutronVision.Simplon.40.dll` — 32–bit version for .Net Framework 4.0
- `LeutronVision.Simplon.40.x64.dll` — 64–bit version for .Net Framework 4.0

This DLL is a wrapper - you can keep it with your application in order not to be forced to recompile it when Simplon libraries are updated to a new version. The backward compatibility is kept on the PlainC API of Simplon (exposed from `lv.simplon.dll`), around which the .Net Class Library wraps, so if the `lv.simplon.dll` changes to a newer version, the old wrapper will still work.

2.2.3.2. Ref versus Out parameters

Functions, which used a pointer to a variable in order to return a value in the PlainC API, are in the NetClass using a parameter *passed by reference*.

The C# distinguishes between "**ref**" and "**out**" - the first is used for in-out parameters, while the second for the out-only parameters. Although Simplon mostly uses such parameters as out-only, the parameters in Simplon methods are declared as "**ref**", because the "**out**" attribute is not supported by C++.

For this reason the C# compiler may issue an error about a non-initialized variable (error CS0165: Use of unassigned local variable 'DeviceId'), for example in this case:

```
String DeviceId;
pInterface.FindDevice(LvFindBy.GevIPAddress, "10.0.2.1", ref DeviceId);
```

Although we know the `DeviceId` is used only for output (returning the value), to satisfy the compiler, the value must be initialized:

```
String DeviceId = "";
pInterface.FindDevice(LvFindBy.GevIPAddress, "10.0.2.1", ref DeviceId);
```

2.2.3.3. Enumerations in the .Net Class Library

The enumerations are derived from the PlainC API by the following rules:

(1) The prefix by the enumeration values is cut off - the identically named values from different enum classes do not clash in .Net Framework:

Plain C, C++ Class Library:

```
enum LvFtrType
{
    LvFtrType_Integer,
    LvFtrType_Float,
    LvFtrType_String,
    ...
};
```

Usage: `LvFtrType_Integer`

.Net Class Library:

```
public enum class LvFtrType : LvEnum
{
    Integer,
    Float,
    String,
    ...
};
```

Usage:

- `LvFtrType::Integer` - C++ .Net
- `LvFtrType.Integer` - C# .Net
- `LvFtrType.Integer` - VB .Net

(2) To avoid a clash with a class name, the feature enumerations are decorated with "`Ftr`" suffix (`LvDevice::DeviceModelName` would clash with the `LvDevice` class, so the `LvDeviceFtr::DeviceModelName` is used):

Plain C, C++ Class Library:

```
enum LvDeviceFtr
{
    LvDevice_DeviceVendorName,
    LvDevice_DeviceModelName,
    LvDevice_DeviceManufacturerInfo,
    ...
};
```

Usage: `LvDevice_DeviceModelName`

.Net Class Library:

```
public enum class LvDeviceFtr : LvFeature
{
    DeviceVendorName,
    DeviceModelName,
    DeviceManufacturerInfo,
    ...
}
```

Usage:

- `LvDeviceFtr::DeviceModelName`- C++ .Net
- `LvDeviceFtr.DeviceModelName` - C# .Net
- `LvDeviceFtr.DeviceModelName` - VB .Net

Unfortunately the .Net Class Library cannot export typedefs, so the `LvFeature` and `LvEnum` typedefs are exported as `UInt32`. Furthermore, C++ and C# requires an annoying explicit retype, even if the enum is internally based on the same type:

```
Device.SetEnum (LvDeviceFtr.PixelFormat, (UInt32)LvPixelFormat.BayerGR8);
```

Visual Basic does not care.

2.2.4. Running multiple instances of application using Simplon

It is possible to run simultaneously multiple applications linked with Simplon and accessing supported hardware. However, there are limitations to be kept on mind:

- Single hardware usually cannot be accessed by multiple applications, for example 2 applications cannot access the same GigE camera. There exists some exceptions — in some cases a special mode is possible, in which the camera can be controlled by one application and another application can act as a passive receiver of images from the same camera; however such mode must be supported by the camera.
- On the *CheckSight* camera the hardware can be accessed by a single application instance only, even in case of dual head models, so please avoid running multiple instances of Simplon application there.

2.2.5. Using debug mode

When you compile the application using Simplon in Visual Studio in the debug mode, the time needed for opening the camera might be significantly longer. This is mostly caused by the XML parser, which decodes the XML describing the remote device, as well as the XMLs of the System, Interface, etc. This parser makes quite a lot of small memory allocations — Visual Studio in debug mode makes the allocations in very complex way in order to be able to diagnose memory leaks and invalid accesses.

Simplon for Windows can be installed with PDB files, which enable to do debugging of Simplon DLLs. This can be very useful for technical support, namely when you experience any kind of crash.

2.2.5.1. Debugging with a GigE camera

The GigE cameras have embedded a mechanism, which assures that when a connection is lost, the camera returns to an unlocked state and can be connected again. This mechanism expects that the application sends to the camera a special packet periodically with a defined frequency (so called heartbeat). When the camera does not receive this in a certain timeout (usually 3 seconds), the connection is evaluated as lost. This also happens, when you are debugging your application and stop it on a breakpoint — the breakpoint stops all the application threads, so if you do not continue from the breakpoint in short time, the connection is lost. Thus it is recommended either to use logging instead of breakpoints, or to increase the heartbeat interval in Simplon Settings.

2.2.6. The documentation of APIs

As there are 3 APIs available, a question arises, in which API to show code snippets in this manual. As the C++ API and the .Net Framework Class Library have very similar architecture, we use the **C++ API as a reference**. You can easily derive the corresponding calls in the other API, for example:

```
// C++ code:
  pRenderer->DisplayImage(pBuffer);
  pBuffer->Queue();

// Plain C code:
  LvRendererDisplayImage(hRenderer, pBuffer);
  LvBufferQueue(hBuffer);
```

As you can see, in the Plain C the name of the module (Renderer, Buffer, Device, ...) is a part of the function name, while in the C++ it is the name of the class. This rule does not apply only to

feature handling functions — as these are common for all modules, the module name is omitted from the function name:

```
// C++ code:
  pDevice->GetInt32(LvDevice_Width, &Width);
  pDevice->GetInt32(LvDevice_Height, &Height);

// Plain C code:
  LvGetInt32(hDevice, LvDevice_Width, &Width);
  LvGetInt32(hDevice, LvDevice_Height, &Height);
```

The `Lv` prefix is used on all published definitions, constant and functions, in order to avoid name clashes with other libraries.

2.2.7. Compiling an old source code with a new Simplon version

The Simplon library is evolving and thus can happen that some features or functions may become obsolete. While such items will be still valid for already compiled application (in order to keep the backward compatibility), they can be conditionally excluded from the header files, just to assure the newly designed applications do not use them. The exclusion is made with the `SIMPLON_INC_OBSOLETE` define, for example

```
#ifdef SIMPLON_INC_OBSOLETE
 LvDevice_LvSensorExposureMode =  36 | LV_DEV_RFTR,
#endif
```

So if you need to recompile an older application with a new Simplon version, you might need to define `SIMPLON_INC_OBSOLETE` in your project, if some previously existing items are not found during the compilation.

2.3. Building an application with Simplon

2.3.1. Error handling

With only a few minor exceptions, every Simplon API function returns a status value, indicating the success of the operation. The success is indicated by a value zero (`LVSTATUS_OK`) The status value can be converted to a string message using the `LvGetErrorMessage()` function. Here is an example:

```
LvStatus Status;
Status = m_pDevice->AcquisitionStart();
if (Status != LVSTATUS_OK)
{
    char Msg[1024];
    LvGetErrorMessage(Status, Msg, sizeof(Msg));
    // display the Msg
}
```

The status values are listed in the `lv.simplon.status.h` file.

The `LvGetErrorMessage()` function simply convert the error number to a string, using an internal static table of error messages. You can obtain more detailed information, when you use the `LvGetLastErrorMessage()` function. This function has a different implementation: for each thread it stores diagnostic info for the last error which occurred. Besides the error description from `LvGetErrorMessage()` this info contains also the name of the function, in which the error happened, and if the error is connected with some feature, it also adds the feature name. In some cases it additionally records some more diagnostic information. The `LvGetLastErrorMessage()` always holds the *last error* description, for speed reasons the message is not cleared if an API function returns `LVSTATUS_OK`. Thus you should call this function only in case you detect an error status in the function return value, similarly as was shown in the previous example:

```
LvStatus Status;
Status = m_pDevice->AcquisitionStart();
```

```
if (Status != LVSTATUS_OK)
{
    char Msg[1024];
    LvGetLastErrorMessage(Msg, sizeof(Msg));
    // display the Msg
}
```

It is important to call the `LvGetLastErrorMessage()` from the same thread, from which was called the API function, which returned the error status, as the diagnostic information is stored separately for each thread.

Hint: if you are troubleshooting some non functional code, it might be also useful to see the Simplon Log — all the errors are recorded there. See Section 3.6, “Simplon log output” [p. 68] for more details.

2.3.1.1. Error handling in the C++ Class Library

All the methods in the C++ Class Library return the error status in the same way as the functions in the plain C API. Besides this, the C++ Class Library provides a possibility to throw an exception when a method is about to return an error value. By default this is switched **off** (in contrast to the .Net Class Library) and you must switch it on explicitly by the `LvLibrary::SetThrowErrorEnable(true)` call.

The exceptions can be optionally of two types — either using the `LvException` class, defined in the C++ Class Library, or using the `exception` class from the standard library. In the latter case, your application must define `LV_USE_STDEXCEPTION` preprocessor variable.

Example:

```
try
{
    m_pDevice->AcquisitionStart();
    // ... and more Simplon API calls, without checking the return value
}
catch (LvException e)
{
    DisplayErrorMsg(e.Message(), e.Number());
    return;
}
```

2.3.1.2. Error handling in the .Net Class Library

By default, the error states are converted to exceptions of the `LvException` class type. Thus, no error handling option is available in the source code generator. Exception handling is shown in the samples generated by the Source Code Wizard.

If you do not like exceptions, you can switch this conversion off and use the classic checking of method return values:

```
LvLibrary.ThrowErrorEnable = false;
```

2.3.2. Opening and closing the library

Before your application uses any of the Simplon functions, it must as the very first action call the `LvOpenLibrary()` function. Conversely, it must call the `LvCloseLibrary()` as the last used Simplon function, usually before the application exit.

These 2 functions serve to initialization and uninitialization of the library. Their calls must be done *during the normal application execution*, it is forbidden to place them for example in the `DllMain()` function of a Windows DLL, because at such place huge code execution restrictions are active.

The only function, which can be called outside the Open/CloseLibrary is the `LvGetVersion()` function.

2.3.2.1. Opening a system

The **System module represents a GenTL producer library**. Simplon comes with single GenTL producer and if you want to use this one, you can simply use an empty string to open it:

```
LvSystem* pSystem;
if (LvSystem::Open("", pSystem) != LVSTATUS_OK)
{
    ErrorMsg("Opening the system failed.");
    return;
}
```

However, if you want to use other than Simplon GenTL producer, you need to specify its ID. The ID is a *file name* of the GenTL producer library. The file name can be specified either with the full path, or without a path; in the latter case the library is searched in the default path for the GenTL libraries, which is set in the operating system.

You can also iterate through available systems using the following functions: `LvUpdateSystemList()`, `LvGetNumberOfSystems()` and `LvGetSystemId()`. This is suitable for example in a case you want to create a menu with all systems available.

Note the way of opening the system in the C++ API: instead of direct using the `new` operator, you are forced to use the way shown in the example above (the constructor is private). The reason for this is a cleanup, when the opening fails — in such case the returned pointer is NULL and thus an unopened system cannot be represented by a valid `LvSystem` class instance. Furthermore, you get the function returns the error status, which you could not get, when using the `new` operator. A similar strategy is used for closing the system — instead of the `delete` operator, you must use the following construct:

```
    LvSystem::Close(pSystem);
```

Note that the `pSystem` pointer is passed by reference and the `Close()` function sets its value to NULL.

The same principle is applied to all Simplon modules (Interface, Device, Stream, ...)

Note: The GenTL does not permit to open the system multiple times. This is quite unpractical, so Simplon enables to open the same system several times in the same process. It does internal reference counting, so the application must balance the number of *opens* with the same number of *closes*; the system is really closed only when the reference count decreases to 0. The same mechanism is also available by the Interface.

2.3.3. The LvSystem module

2.3.3.1. Opening an interface

The System module provides one or more interfaces. The Interface module represents one physical interface in the system. For Ethernet based devices this would be a Network Interface Card (NIC); for a Camera Link based implementation this would be one frame grabber board. One system may contain one or multiple interfaces and you can enumerate them. By the `LvSystem::UpdateInterfaceList()` you ask the system to update the internal interface list according current hardware status. By the `LvSystem::GetNumberOfInterfaces()` and `LvSystem::GetInterfaceId()` you can obtain a list of interfaces (their IDs). Once you know the interface ID, you can open it, using the `LvSystem::OpenInterface()` function and close it using the `LvSystem::CloseInterface()`. The following code opens and closes all found interfaces — this code does not have a practical usage, but it illustrates well how to iterate through the existing interfaces, obtain their IDs open and close them:

```
uint32_t NumberOfInterfaces;
pSystem->GetNumberOfInterfaces (&NumberOfInterfaces);
for (uint32_t i=0; i<NumberOfInterfaces; i++)
```

```
{
    std::string sInterface;
    pSystem->GetInterfaceId(i, sInterface);
    LvInterface* pInterface;
    if (pSystem->OpenInterface(sInterface.c_str(), pInterface)
        != LVSTATUS_OK)
    {
        // display error message
        continue;
    }
    // do something with the interface
    pSystem->CloseInterface(pInterface);
    // CloseInterface() sets the pInterface pointer to NULL
}
```

The code above is useful to build a menu of interfaces, but this is usually not what is needed in a real application — there you rather need to select the appropriate interface automatically. The simplest solution — using a hardcoded ID — may cause problems when it comes to a maintainability of such application: The string ID of the interface is a *unique* identifier, thus you must count with the possibility, that when you for example exchange a defective NIC by another piece, the ID of the interface might change (similar situation is with the Device ID). The chapter Section 2.5, “Writing maintainable applications” [p. 57] discusses this topic in detail and provides hints how to solve this issue. Also, some GenTL providers enable using alternative IDs, like an *IP address* or a *nickname* (User ID) for opening the Interface or Device.

As already mentioned by the System, the `LvInterface` constructor and destructor are private and thus you must use the *Open* and *Close* functions to create the `LvInterface` class instance. As the Interface has always an owner System, the *Open* and *Close* functions are available as methods of `LvSystem`:

```
LvStatus LvSystem::OpenInterface (const char* pInterfaceId,
                                  LvInterface*& pInterface);
LvStatus LvSystem::CloseInterface (LvInterface*& pInterface);
```

But this just for your convenience, these methods in fact call the static `LvInterface` class methods, which have the same form as `LvSystem` *Open*/*Close* functions.

```
static LvStatus LvInterface::Open (LvSystem* pSystem,
                                   const char* pSystemId,
                                   LvInterface*& pInterface);
static LvStatus LvInterface::Close (LvInterface*& pInterface);
```

The same principle is applied to all other modules — the owner module provides methods for opening and closing the owned modules for example the `LvInterface` class provides methods for opening and closing the *Device* module.

Note: The GenTL does not permit to open the interface multiple times. This is quite unpractical, so Simplon enables to open the same interface several times in the same process. It does internal reference counting, so the application must balance the number of *opens* with the same number of *closes*; the interface is really closed only when the reference count decreases to 0.

2.3.3.2. Getting info about an interface

You can query for some interface characteristics (for example its IP address) without opening it. These characteristics are available through the `LvModule::GetInfo()` function with one of the `LvFtrInfo_Interface`*xxx* constants. Alternatively, you can also use the System *features* under the `LvSystem_InterfaceSelector` to iterate through interfaces and check the interface features (see Section 2.4, “Features” [p. 50]). A sample source code for this is shown in the Section 2.5, “Writing maintainable applications” [p. 57].

2.3.4. The LvInterface module

The Interface module provides devices. The typical device is a camera (in the GenICam terminology it is called a *device*, because it can also be a scanner or any other device communicating in defined way).

2.3.4.1. Opening a device

Similarly as with Interfaces, you can enumerate and open Devices. The following code again does not have practical usage, but it illustrates how to iterate Devices, obtain their IDs, open and close them.

```
uint32_t NumberOfDevices;
pInterface->GetNumberOfDevices(&NumberOfDevices);
for (uint32_t i=0; i<NumberOfDevices; i++)
{
    std::string sDevice;
    pInterface->GetDeviceId(i, sDevice);
    LvDevice* pDevice;
    if (pInterface->OpenDevice(sDevice.c_str(), pDevice)
        != LVSTATUS_OK)
    {
        //ErrorMsg("Opening the device failed.");
        continue;
    }
    // do something with the interface
    pInterface->CloseDevice(pDevice);
    // CloseDevice() sets the pDevice pointer to NULL
}
```

The Device ID is also a unique identifier, which differs from piece to piece, so it is important (even more than by the Interfaces) to select the proper way, how to obtain it. The chapter Section 2.5, “Writing maintainable applications” [p. 57] discusses ways how to obtain the ID, so that the application is well maintainable.

2.3.4.2. Getting info about the device

You can query for some device characteristics (for example its IP address) without opening it. These characteristics are available through the `LvModule::GetInfo()` function with one of the `LvFtrInfo_Device`*xxx* constants. Alternatively, you can also use the Interface*features* under the `LvInterface_DeviceSelector` to iterate through devices and check the device features (see Section 2.4, “Features” [p. 50].). A sample source code for this is shown in the Section 2.5, “Writing maintainable applications” [p. 57].

2.3.4.3. The device access

When you open the Device, you can specify the as the last parameter to the `LvInterface::OpenDevice()` function the access mode. By default it is `LvDeviceAccess_Exclusive`, which means once the device is opened, no other thread or application can access it. Some types of cameras enable sharing the device by multiple clients, in such case one client is controlling the camera (opens the camera with the `LvDeviceAccess_Control` mode) and all the other clients can only receive data from this device (open the camera with the `LvDeviceAccess_ReadOnly` mode). Note that if this sharing is not supported by the device, the opening in the specified access mode will fail.

2.3.5. The LvDevice module

The Device module represents the acquisition device, typically a camera. Usually, it provides a rich set of *features*, by which it can be controlled and configured. The features are discussed in detail in the chapter Section 2.4, “Features” [p. 50].

2.3.5.1. Opening a stream

The Device delivers the image data (or other data) in form of a stream; it can provide multiple types of streams. You can enumerate the available streams by the `LvDevice::GetNumberOfStreams()` and `LvDevice::GetStreamId()`, however, in contrast to finding the devices, the situation with streams is much simpler: Most cameras provide only one stream, so you can simply use the ID of the stream at index 0. If some camera provides more streams, it is still most likely that the first stream is the image buffers stream and any special data streams are on the higher indexes. The additional streams are anyway always documented in the manual of the particular camera (device).

Simplon enables to use an empty string instead of the stream ID in the `LvDevice::OpenStream()`, this frees you from the need to obtain the stream ID. In such case the first found stream is used.

```
pDevice->OpenStream("", pStream);
```

2.3.6. The LvStream module

The stream provides the image data in buffers. The buffers can be either allocated by Simplon or by the user application. In any case, the software in lower layers might need to do additional actions before starting the acquisition (typically, the driver must lock the memory belonging to the buffers, in order to be able to obtain their physical addresses and use the DMA transfers). Thus it is not permitted to change the number of buffers or reallocate them during the acquisition.

2.3.6.1. Allocating buffers

It is up to the application how many buffers it needs to use. Usually it is more than one; while the application inspects the contents of the acquired image, the acquisition can continue to another buffer. A stream defines a minimum number of buffers needed for the acquisition (see the `LvStream_StreamAnnounceBufferMinimum` feature). A maximum is not limited in Simplon, it is rather done by the HW configuration — usually there must be enough physical memory available to hold all buffers (cannot be swapped to HDD) and still leave reasonable space for the operating system.

A buffer is represented by the LvBuffer module in Simplon. Here is the easiest way how to allocate the buffers (no error handling included):

```
LvBuffer* Buffers[NUMBER_OF_BUFFERS];
for (int i=0; i<NUMBER_OF_BUFFERS; i++)
{
    pStream->OpenBuffer(NULL, 0, NULL, 0, Buffers[i]);
}
```

The first parameter (`pDataPointer`) of the `LvStream::OpenBuffer()` function is a pointer to the buffer data . If this is NULL, the buffer is allocated by Simplon.

The second parameter (`DataSize`) specifies the size of the data . If this is 0, Simplon automatically determines the size of the buffers to be allocated.

The third parameter (`pUserPointer`) provides a possibility to attach a pointer to each buffer (which can be for example used to link the buffer with some other internal structure or class), we do not use this in this sample.

And the fourth parameter are optional flags, not used in this sample.

In case you want to allocate the buffers in your application, you first need to determine the size of the buffers needed. You should keep in mind that the size is not affected only by the image size and its pixel format, but also by other factors: typically, when you switch on chunk data (data attached to each image, holding additional information, like a Frame ID or timestamp), the required size is larger, in order to be able to hold these attached data. The size needed is referred as *Payload*

size and Simplon Stream module provides a feature for obtaining it. The following code snippet shows allocation and deallocation of buffers in the application:

```
LvBuffer* Buffers[NUMBER_OF_BUFFERS];

// allocate buffers
int32_t PayloadSize;
pStream->GetInt32(LvStream_LvCalcPayloadSize, &PayloadSize);
for (int i=0; i<NUMBER_OF_BUFFERS; i++)
{
    void* pData = malloc(PayloadSize);
    pStream->OpenBuffer(pData, PayloadSize, NULL, 0, Buffers[i]);
}

// free buffers
for (int i=0; i<NUMBER_OF_BUFFERS; i++)
{
    void* pData;
    Buffers[i]->GetPtr (LvBuffer_Base, &pData);
    pStream->CloseBuffer(Buffers[i]);
    free(pData);
}
```

It is **important to keep in mind**, that changing the device configuration (its features) might result in the change of the payload size. If the buffers are allocated with an insufficient size, the acquisition start will fail. So in your application is necessary either to assure the configuration is done before the buffers are allocated, or, if it is a user GUI application like Simplon Explorer, recheck before each acquisition start, if the payload size did not change — and if so, close all buffers and allocate them again. This applies also to the case when the buffer allocation is done by Simplon — Simplon checks the payload size at the moment of the `LvStream::OpenBuffer()` call, not later.

2.3.6.2. Input buffer pool, output queue

The principle of the image acquisition is that Simplon fills in the prepared buffer with the image data and passes it to the application. The application processes the buffer and after it is done with it, it passes it back to Simplon, so that it can be reused for the acquisition of another image. Let's have a look at this mechanism in detail.

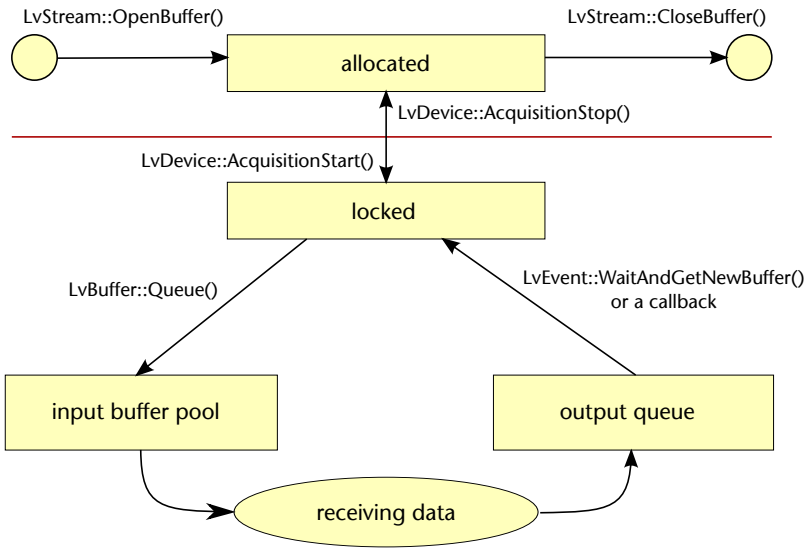


Figure 2.1. Simplon buffers

Before starting the acquisition all the buffers are *Allocated*. At this state it is possible to add more buffers by `LvStream::OpenBuffer()` or delete buffers by `LvStream::CloseBuffer()`.

When the acquisition starts, all buffers are *Locked*, that means the application no longer can change their number or size. Each buffer has one of the following 3 ownerships:

- Owned by the **input buffer pool**.
- Owned by the **output buffer queue**.
- Owned by the **application**.

The **input buffer pool** is a stock of free buffers. Note the usage of the word *pool* instead of *queue*. This means that the order in which the free buffers are filled is not defined, it is not guaranteed that the last buffer put into the pool will be filled last. In other words, once the application puts the buffer to the input buffer pool, it should not access it anymore.

When a new image is being acquired, one buffer is taken from the input buffer pool and is filled. When it is complete, the buffer is passed to the **output buffer queue**. From this queue the application picks up the buffers, in the order they were acquired.

The application puts the buffer to the input buffer pool by the `LvBuffer::Queue()` function call.

To get the buffer from the output buffer queue, the application either can use the `LvEvent::WaitAndGetNewBuffer()` function or it can configure Simplon to be notified by a *callback* function (this is discussed in detail in the chapter Section 2.3.8, “The LvEvent module” [p. 43]).

The figure shows that before the application can start the acquisition, it must put sufficient number of buffers to the input buffer pool, using the `LvBuffer::Queue()` function.

Sometimes it is needed to move the buffers from one owner to another. The `LvStream::FlushQueue()` function is providing such actions. The following figure shows possible movements and corresponding parameters to the `LvStream::FlushQueue()` function.

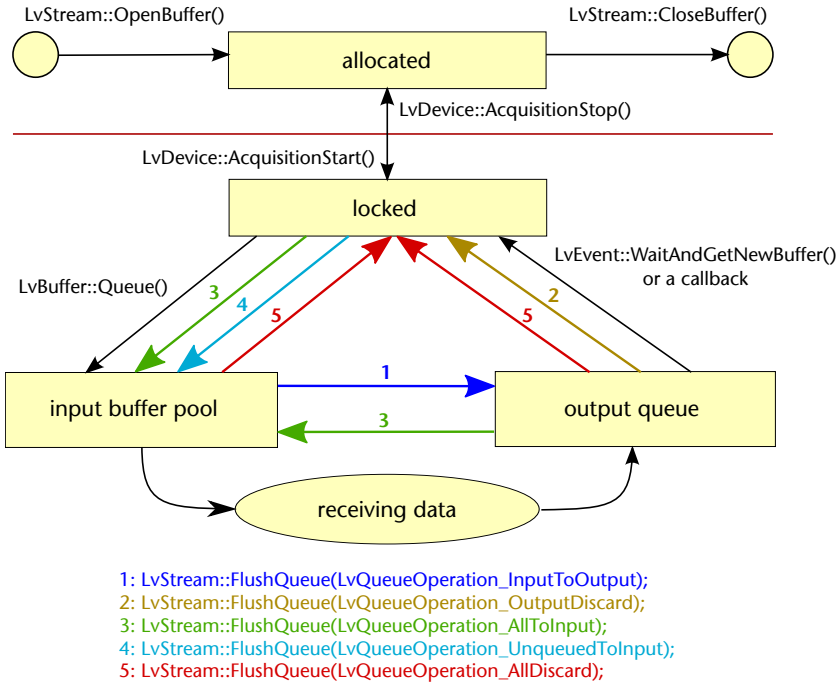


Figure 2.2. Flushing Simplon buffers

2.3.7. The LvBuffer module

The Buffer module represents one buffer. To make a unified API , the buffer is also providing *features* (see Section 2.4, “Features” [p. 50]), however, internally the buffer features are implemented by special means, assuring that each buffer does not consume a lot of additional memory just for handling a standard XML tree of features.

In the application, you do not need to keep a list of buffers; it is always kept in Simplon, so the code for deleting the buffers from the previous chapter could be written alternatively like this (no Buffers[] array used):

```
// free buffers
int32_t Count;
pStream->GetInt32(LvStream_StreamAnnouncedBufferCount, &Count);
for (int i=Count-1; i>=0; i--)
{
    LvBuffer* pBuffer;
    pStream->GetBufferAt(i, pBuffer);
    void* pData;
    pBuffer->GetPtr (LvBuffer_Base, &pData);
    pStream->CloseBuffer(pBuffer);
    free(pData);
}
```

Note the loop going from `Count-1` down to `0` instead of vice versa; this is necessary because every `LvStream::CloseBuffer()` decrements the number of buffers, so when going vice versa, the index `i` would become invalid at the half of the range.

Sometimes the requirement can be vice-versa: The application needs to keep an additional information with each buffer. For this purpose the buffer module can be assigned a *user pointer*, which is held in the module and is available when the buffer needs to be processed. The user pointer is passed as a parameter in the `LvStream::OpenBuffer()` and can be read back, when the buffer is for example received from the output buffer queue. Note that this is one of the few cases, where the Plain C API differs from the C++ API. In the Plain C API the call to obtain the user buffer is the following call:

```
LvGetPtr (hBuffer, LvBuffer_UserPtr, &pUserPtr);
```

However, in the C++ API the original user pointer is internally used to hold the pointer to the `LvBuffer*` class instance, so the actual user buffer is kept as a member in the `LvBuffer` class and is available in the `LvBuffer::GetUserPointer()` function:

```
pUserPtr = pBuffer->GetUserPtr();
```

2.3.8. The LvEvent module

The Event module is providing a mechanism for informing the application about asynchronous events, typically about a new buffer being put to the output buffer queue. It also provides an optional thread, which waits for the events and upon them calls the application defined callback function; the application then does not need to deal with maintaining an additional thread and its source code can be identical for multiple operating systems.

Depending on the event type, the Event module is created and owned by System, Device or Stream.

2.3.8.1. Types of events

There are 3 types of events:

- `LvEventType_NewBuffer` — this is the main event type, it informs about new buffer was put to the output buffer queue and thus is ready to be taken over by the application. Can be opened only by the Stream module.

- `LvEventType_FeatureDevEvent` — this event informs about a feature on a Device, which has changed. The event itself is processed internally and converted to a feature callback (see `LvEvent::RegisterFeatureCallback()`), but it needs an additional thread to be started for processing it and this must be done explicitly by the application. This event type is discussed in detail in the chapter Section 3.7.2.1, “Capturing logs from PicSight GigE Smart” [p. 73]. Can be opened only by the Device module.

- `LvEventType_Error` — this type of event is used to signalize an error occurred in another thread. It is discussed in detail in *Advanced topics*. Can be opened by the System, Device and Stream modules.

2.3.8.2. Waiting for an event

The nature of the events is asynchronous, that means the events happen independently on the application main thread run. The function `LvEvent::WaitAndGetData()` provides waiting for the event and extracting the data from the event queue in one atomic operation. For the `LvEventType_NewBuffer` type of the Event, this function has the form `LvEvent::WaitAndGetNewBuffer()` — in contrast to the `LvEvent::WaitAndGetData()`, this function provides directly the pointer to obtained `LvBuffer` class. After the return from this function, the Buffer is extracted from the output queue and is owned by the application. After the application is finished with the buffer, it must return it to the input buffer pool, using the `LvBuffer::Queue()` function.

The above mentioned functions for waiting for the event are *blocking*, that means the function does not return until the event comes or the timeout expires. For this reason, the processing of the events is usually not done in the main application thread — this would make the application non responsive while it is blocked by waiting for an event. Instead of this, an additional thread is usually run, which processes the events in a loop. The typical loop looks as follows:

```
bool Terminated = false;
while (!Terminated)
{
    LvBuffer* pBuffer;
    pEvent->WaitAndGetNewBuffer(pBuffer);
    if (Terminated) break;
    // get the pointer to the image data
    void* pData = NULL;
    pBuffer->GetPtr (LvBuffer_Base, &pData);
    // process the buffer data...

    // ... and finally return it back to the input buffer pool
    pBuffer->Queue();
}
```

As you can see ate the end of the code snippet, the Buffer is returned to the input buffer pool after it is processed. This is an important thing in the buffer handling, which you must not forget in the code.

As already mentioned, the `LvEvent::WaitAndGetNewBuffer()` is blocking. To be able to break the loop even if the thread is just inside this function, you need to utilize the `LvEvent::Kill()` function, which terminates the waiting for the event:

```
Terminated = true;
pEvent->Kill();
```

2.3.8.3. Using the callback function

Simplon offers a more convenient way how to obtain the events: when the application supplies a *callback function*, Simplon can start an internal thread, which runs an internal loop with the `LvEvent::WaitAndGetNewBuffer()` and calls the callback function for each obtained event. This approach simplifies the application code and enables to use the same code in different operating system.

All what is needed is to register the callback function, start the thread before starting the acquisition (one function call), and stop it after the acquisition stop.

This mechanism also provides an optional possibility to start only the thread, without specifying the callback function — in such case each acquired image is displayed (if the Renderer module is configured) and automatically passed back to the input buffer pool.

The callback function has the same form in both the Plain C and C++ API:

```
void LV_STDC LvEventCallbackNewBufferFunct (LvHBuffer hBuffer,
                                            void* pUserPointer,
                                            void* pUserParam);
```

The first parameter is a buffer handle, which is used in the Plain C API. In the C++ API is the pUserPointer parameter utilized to hold the pointer to the `LvBuffer` class instance, representing the buffer. The third parameter is user parameter, which can be utilized by the application to identify callbacks from various sources.

In the following code snippet we have a `CCamera` class, which encapsulates the functionality for an acquisition on a single device. We will write the code so that this class could be used in multiple instances, each for one device. As the callback function cannot be a method of the class, we utilize the `pUserParam` to identify for which `CCamera` class instance this callback belongs — then we do not need to have multiple callback functions. The function simply identifies the `CCamera` instance and calls its `CallbackNewBuffer()` method:

```
void LV_STDC CallbackNewBufferFunction(LvHBuffer hBuffer,
                                       void* pBuffer,
                                       void* pUserParam)
{
    CCamera* pCamera = (CCamera*) pUserParam;
    pCamera->CallbackNewBuffer((LvBuffer*) pBuffer);
}
```

The callback function is to be registered by the `LvEvent::SetCallbackNewBuffer()` function. Below is the typical code snippet for opening the Interface, Device, Stream and Event and setting the callback function. Note the `this` parameter in the `LvEvent::SetCallbackNewBuffer()` — this is the pointer, which is then passed as the `pUserParam` to the callback function, so we can identify to which `CCamera` class instance the callback belongs. Note also that as usual, the error handling is omitted for simplicity in this code; in a real application the calls like `OpenDevice()` should always count with a possibility of failure (for example when the camera is locked by another application), so the error handling should be always used.

```
// in the CCamera class declaration:
LvInterface* m_pInterface;
LvDevice*    m_pDevice;
LvStream*    m_pStream;
LvEvent*     m_pEvent;

void CCamera::OpenCamera(LvSystem* pSystem,
                         std::string sInterface,
                         std::string sCameraId)
{
    pSystem->OpenInterface(sInterface.c_str(), m_pInterface);
    m_pInterface->OpenDevice(sCameraId.c_str(), m_pDevice);
    m_pDevice->OpenStream("", m_pStream);
    m_pStream->OpenEvent(LvEventType_NewBuffer, m_pEvent);
    m_pEvent->SetCallbackNewBuffer(CallbackNewBufferFunction, this);
}
```

Before you start the acquisition, start the thread in the Event module, after you stop the acquisition, stop the thread (no error handling shown):

```
void CCamera::StartAcquisition()
{
    m_pEvent->StartThread();
    m_pDevice->AcquisitionStart();
```

```
}

void CCamera::StopAcquisition()
{
    m_pDevice->AcquisitionStop();
    m_pEvent->StopThread();
}
```

The processing in the callback function is the same as we shown in the previous chapter:

```
void CCamera::CallbackNewBuffer(LvBuffer* pBuffer)
{
    // get the pointer to the image data
    void* pData;
    pBuffer->GetPtr (LvBuffer_Base, &pData);
    // process the buffer data...

    // ... and return the buffer to the input buffer pool
    pBuffer->Queue();
}
```

Important note: you should always keep in mind that the callback function is called *from a different thread* than is the main application thread. So any functions called from the callback function must be *thread safe*. Typically, problems may arise in GUI environments, where the GUI elements can be accessed only from the main thread. For example in the QT environment you might need to use a `QLabel` widget to display some information for each frame (Frame ID, timestamp, etc.). However, as the `QLabel` instance was created in the main thread, you cannot call directly its `setText()` method from the callback function (= from another thread); instead you will need to utilize a *signal-slot* mechanism (or similar) to deliver the info to the main thread and to update the label in the main thread. Such limitations are quite common in GUI frameworks (similar are also in Windows Forms in MS .Net Framework), and are usually not sufficiently documented.

2.3.8.4. Events in .Net Class Library

In the Simplon .Net Class Library the callbacks are converted to .Net Framework events. The handling of the events is the following (it is also demonstrated in the samples generated by the Source Code Wizard):

C++:

1) Create the handler in this form:

```
void CCamera::NewBufferEventHandler(System::Object^ sender,
                                    LvNewBufferEventArgs^ e)
{
    try
    {
        IntPtr pData;
        e->Buffer->GetPtr (LvBufferFtr::UniBase, pData);
        // do something with pData

        m_pRenderer->DisplayImage(e->Buffer);
        e->Buffer->Queue();
    }
    catch (LvException^)
    {
        // just log an error, do not display message box here
    }
}
```

2) Assign the handler to the `OnEventNewBuffer` event of the `LvEvent` class instance:

```
m_pEvent->OnEventNewBuffer += gcnew LvEventNewBufferHandler(this,
                                        &CCamera::NewBufferEventHandler);
```

3) Start the callback mechanism inside Simplon:

```
m_pEvent->SetCallbackNewBuffer(true, IntPtr(0));
m_pEvent->StartThread();
```

The `LvEvent::SetCallbackNewBuffer()` call is needed to tell Simplon it should start using the callback (which is then internally converted to the .Net Framework event) and the `LvEvent::StartThread()` call starts an internal thread, which is actually calling the callback.

C#:

1) Create the handler in this form:

```
void NewBufferEventHandler(System.Object sender, LvNewBufferEventArgs e)
{
    try
    {
        IntPtr pData = (IntPtr) 0;
        e.Buffer.GetPtr(LvBufferFtr.UniBase, ref pData);
        // do something with pData

        m_pRenderer.DisplayImage(e.Buffer);
        e.Buffer.Queue();
    }
    catch (LvException)
    {
        // just log an error, do not display message box here
    }
}
```

2) Assign the handler to the `OnEventNewBuffer` event of the `LvEvent` class instance:

```
m_pEvent.OnEventNewBuffer += new LvEventNewBufferHandler(NewBufferEventHandler);
```

3) Start the callback mechanism in Simplon:

```
m_pEvent.SetCallbackNewBuffer(true, (IntPtr)0);
m_pEvent.StartThread();
```

The `LvEvent::SetCallbackNewBuffer()` call is needed to tell Simplon it should start using the callback (which is then internally converted to the .Net Framework event) and the `LvEvent::StartThread()` call starts an internal thread, which is actually calling the callback.

Visual Basic:

1) Declare the `LvEvent` instance with the `WithEvents` keyword:

```
Private WithEvents m_pEvent As LvEvent
```

2) Create the handler in this form:

```
Public Sub NewBufferEventHandler(ByVal sender As System.Object, _
                                 ByVal e As LvNewBufferEventArgs) _
                                 Handles m_pEvent.OnEventNewBuffer
    Try
        Dim pData As IntPtr
        e.Buffer.GetPtr(LvBufferFtr.UniBase, pData)
        '  do something with pData

        m_pRenderer.DisplayImage(e.Buffer)
        e.Buffer.Queue()
    Catch ex As LvException
        ' just log an error, do not display message box here
    End Try
End Sub
```

You can see the assignment of the handler to the `OnEventNewBuffer` event of the LvEvent class instance is done directly in the function definition.

3) Start the callback mechanism in Simplon:

```
m_pEvent.SetCallbackNewBuffer(True, 0)
m_pEvent.StartThread()
```

The `LvEvent::SetCallbackNewBuffer()` call is needed to tell Simplon it should start using the callback (which is then internally converted to the .Net Framework event) and the `LvEvent::StartThread()` call starts an internal thread, which is actually calling the callback.

2.3.9. Running the acquisition

The start of the acquisition involves multiple actions, which are encapsulated in the `LvDevice::AcquisitionStart()` function.

- The buffers are checked for size and number. If the size of some buffer is less than the required payload size, or the number of buffers is less than the minimum required, the `LvDevice::AcquisitionStart()` returns an error status.
- The `TLParamsLocked` feature on the device is set to true. This is a special feature, which disables changing some of the features during the acquisition (like the image size and pixel format).
- The `LvStream::Start()` is called, if it was not already called by the application.
- The `AcquisitionStart` command feature is executed on the remote device.

The process of stopping the acquisition is encapsulated in the `LvDevice::AcquisitionStop()` function.

- The `AcquisitionStop` command feature is executed on the remote device.
- The `LvStream::Stop()` is called, if the `LvStream::Start()` was called from the `LvDevice::AcquisitionStart()`.
- The `TLParamsLocked` feature on the device is set to false.

There might be special cases, when some of the actions above is to be skipped or done differently; these exceptions are available by the Options parameter of the `LvDevice::AcquisitionStart()` and `LvDevice::AcquisitionStop()` function (discussed in advanced topics). But for most cases these functions can be used without any special options.

Note: As you can see from the above, starting the acquisition is not equal to the call of the `AcquisitionStart` command feature on the remote device. Thus the `AcquisitionStart`, `AcquisitionStop`, `AcquisitionAbort` and `AcquisitionArm` features are not represented by a constant in Simplon API, as their direct use from the application is not expected.

2.3.9.1. Aborting the acquisition

Sometimes it might be needed to abort pending acquisition. This functionality is provided by the `LvDevice::AcquisitionAbort()` function. It sends to the remote device a `AcquisitionAbort` command and then calls the `LvDevice::AcquisitionStop()`.

2.3.9.2. Preparing for the acquisition

On some devices the preparation for the acquisition can take a relatively long time. This time would be normally spent in the `LvDevice::AcquisitionStart()` function. However, you can call the `LvDevice::AcquisitionArm()` to tell the device to prepare for the acquisition; then this initialization time is moved to this function and the `LvDevice::AcquisitionStart()` then takes less time (starts the acquisition immediately). The `LvDevice::AcquisitionArm()` also includes a start of the stream, if it was not yet done by the application.

2.3.10. The LvRenderer module

The Renderer module is Simplon add-on for an easy display of images. Its functionality is implemented in a standalone library, which is loaded only when you attempt to use the Renderer module in your application. The reason for this is that the environment for display may not be available; while in Windows it is available permanently, in Linux it depends on the presence of the X-Window layer. If the environment is not available, the Renderer module functions simply return error status, but it has no influence to the other Simplon functionality.

The display is implemented using the common operating system functions — in Windows it is `SetDIBitsToDevice()` for unscaled images and `StretchDIBits()` for scaled images; in Linux it is the `XPutImage()` function. Depending on the image pixel format and its line increment, it may be also needed to convert the image to an appropriate format (done automatically). All this process is reasonably fast on modern PCs and is usually sufficient for the display, but you must keep in mind that it may take a significant part of the CPU power. In case this would be a problem, you should consider own image painting by some HW accelerated means (for example using the DirectX in Windows), which are not supported by the Renderer module.

The usage of the Renderer module is quite simple. You create the module and pass to it a handle of the target window:

```
LvRenderer* m_pRenderer;

m_pStream->OpenRenderer(m_pRenderer);
m_pRenderer->SetWindow(hDisplayWnd);
```

In Linux, the `LvRenderer::SetWindow()` has 2 parameters- a pointer to Display and a handle to Window.

Note that the window for displaying the image should be a fully dedicated to the image painting, that means not having any children.

Then the only action needed to display an image is to call the `LvRenderer::DisplayImage()` method, which as the parameter takes a Buffer pointer (C++ version) or Buffer handle (Plain C version). This call is usually put at 2 places:

- At the place, where you obtain the newly acquired buffer — usually the callback function.
- At the repaint handler, which the operating system invokes whenever there is need to repaint the contents of the window. This is discussed in the next chapter.

The code from Section 2.3.8.3, “Using the callback function” [p. 44] would now look like this:

```
void CCamera::CallbackNewBuffer(LvBuffer* pBuffer)
{
    // get the pointer to the image data
    void* pData;
    pBuffer->GetPtr (LvBuffer_Base, &pData);
    // process the buffer data...

    // display the image
    m_pRenderer->DisplayImage(pBuffer);
    // ... and return the buffer to the input buffer pool
    pBuffer->Queue();
}
```

2.3.10.1. Options for painting the image

The Renderer offers several options for displaying the image. These options are available as Renderer *features*. These features include the `RenderType`, which sets the way how the images are to be displayed (clipped, scaled or tiled), offset, size, etc. You can investigate these options in Simplon Explorer.

The typical usage of the `LvOffsetX` and `LvOffsetY` features is for scrolling the image with scroll bars. In Windows the code for image scrolling could look like this:

```
int XPos = GetXScrollPos(hDisplayWnd);
int YPos = GetYScrollPos(hDisplayWnd);
if (XPos != LastXPos || YPos != LastYPos)
{
    LastXPos = XPos;
    LastYPos = YPos;
    m_pRenderer->SetInt(LvRenderer_LvOffsetX, -XPos);
    m_pRenderer->SetInt(LvRenderer_LvOffsetY, -YPos);
}
m_pRenderer->DisplayImage(pBuffer);
```

2.3.10.2. Using the Repaint function

In GUI applications is a common requirement that the application can repaint itself (for example after an overlapping window is removed). The Renderer module provides a simple function for this: `LvRenderer::Repaint()`. This function repaints the last painted buffer and its usage is quite simple — place it in the handler where the repaint is implemented, for example in Windows in the `WM_PAINT` message handler.

But there is one important thing to mention: In order to be able to paint the buffer, the buffer needs to be still in the ownership of the application, that means it must not be returned to the input buffer pool. To assure this, it would mean you could not call `LvBuffer::Queue()` until you are sure you will not need to repaint the image. Simplon offers a comfortable solution for this: it enables *postponing* the return of the buffers to the input buffer queue. The Stream feature `LvStream_LvPostponeQueueBuffers` is designed for this purpose. If you set this value for example to 6, an internal FiFo queue is created for 6 buffers, and each `LvBuffer::Queue()` call passes the buffer to this queue instead of to the input buffer pool. When the seventh buffer is inserted, the first one is passed to the input buffer pool, and so on. This postponing thus assures that the last N buffers are still in the ownership of the application, so if you want to display last 6 images as tiles, they all will be available.

Obviously, the *number of postponed buffers + minimum number of buffers in the input buffer pool* must be less or equal to the number of allocated buffers, otherwise this option can block the acquisition (there would not be enough buffers in the input buffer pool).

2.4. Features

An introduction to features is in the Section 1.5, “Working with features” [p. 18] chapter. The following text describes the features in detail.

2.4.1. Feature groups

There are 3 main groups of features:

- **Remote features** — these features are provided only by the Device module. After the device is opened, it dynamically reports to Simplon its set of features.
- **GenTL features** — these features are provided by the GenTL library for the System, Interface, Device and Stream modules.
- **Local features** — Simplon uses the same mechanism to provide own features; local features are available for System, Interface, Device, Stream and Renderer modules.

As the same set of feature functions is used for each module, in the C++ class API the feature functions are members of the `LvModule` class, from which the LvSystem, LvInterface, LvDevice, etc. classes inherit them. So when referring to these functions, we will use `LvModule::` prefix in the next text.

2.4.2. Obtaining a feature ID

2.4.2.1. Generic use

When you want to work with a feature, you must obtain its ID. The ID is a number which identifies the feature. It is quite similar to a *handle*, the difference is that in place of the ID you can also use a predefined constant, as will be explained later.

To obtain the ID, you must know the feature *name* and *group*. The *name* of the selected feature is visible in Simplon Explorer in the info panel below the tree:

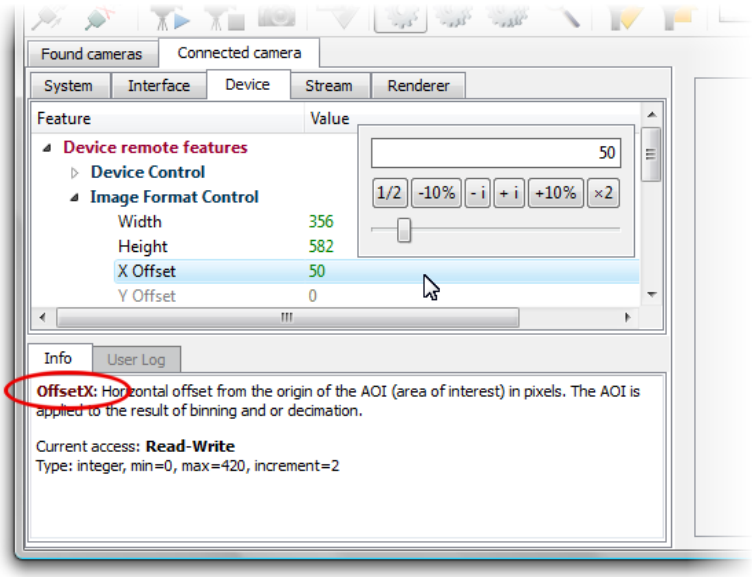


Figure 2.3.

Do not confuse the feature *Name* with the feature *Display name*, which is used in the tree of features.

The *group* is a value from the `LvFtrGroup` enumeration.

You can get the feature ID by the `LvDevice::GetFeatureByName()` function.

```
LvFeature OffsetX_ID;
int32_t OffsetX;
if (pDevice->GetFeatureByName(LvFtrGroup_DeviceRemote,
                              "OffsetX", &OffsetX_ID)== LVSTATUS_OK)
{
    pDevice->GetInt32(OffsetX_ID, &OffsetX);
}
```

This method of obtaining the ID is called **generic**. Its advantage is that you can work even with the features, which were not known to Simplon in the time of its release (new features or features on 3rd party devices). A disadvantage is the complex code for each feature and impossibility to check this code by the compiler — if you make a typing error in the feature name, you will find this error not at compile time, but in the run-time. That is why checking the error status (return value of functions) is always needed.

2.4.2.2. Using a predefined constant

For your convenience, Simplon defines **constants** for all features supported by Leutron Vision hardware. With the constant the code is simpler:

```
int32_t OffsetX;
pDevice->GetInt32(LvDevice_OffsetX, &OffsetX);
```

As you already know, the Simplon Explorer offers source code for each feature get/set in the info panel, so you can easily copy and paste it in your application.

2.4.3. Feature type and GUI

The feature type defines if its value is *boolean*, *integer*, *float*, *string*, *enumeration*, *pointer* or *buffer*. There is also a special feature of the *Command* type. All the types are discussed in the following chapters.

Besides the type, the feature has a recommended GUI representation, for example the `LvFtrGui_IntSlider` is a linear slider. This enables to automatically build a user interface without knowing in advance which features will be included. A feature tree in Simplon Explorer is an example of such user interface.

The type and GUI representation can be obtained by the `LvModule::GetType()` function:

```
LvFtrType FeatureType;
LvFtrGui  FeatureGuiCtl;
pDevice->GetType (LvDevice_OffsetX, &FeatureType, &FeatureGuiCtl);
```

2.4.4. Feature access

The feature can have one of the following access states:

- `LvFtrAccess_NotImplemented` - The feature is defined, but not implemented.
- `LvFtrAccess_NotAvailable` - The feature is implemented, but under the current conditions is not available. The availability can change dynamically upon change of other features. For example the *TriggerSource* device feature becomes available only after the *TriggerMode* feature is set to *On*.
- `LvFtrAccess_ReadOnly` - The feature is available and is read only.
- `LvFtrAccess_WriteOnly` - The feature is available and is write only.
- `LvFtrAccess_ReadWrite` - The feature is available and is fully accessible.

To obtain the access status, use the `LvModule::GetAccess()` function:

```
LvFtrAccess FeatureAccess;
pDevice->GetAccess (LvDevice_OffsetX, &FeatureAccess);
```

For convenience, Simplon provides the following boolean functions for checking the feature access status:

- `LvModule::IsImplemented()` — the feature does not have the *not-implemented* status.
- `LvModule::IsAvailable()` — the feature is *read-only*, *write-only* or *read-write*.
- `LvModule::IsReadable()` — the feature is *read-only* or *read-write*.
- `LvModule::IsWritable()` — the feature is *write-only* or *read-write*.

2.4.5. The Integer feature

The GenICam always works with a 64–bit representation of integer values. Although this is a universal solution, only a few features actually require larger than 32–bit integer. For convenience, Simplon offers working with both 64–bit integers and 32–bit integers (which are internally handled

as 64–bit). To set an integer value, use the `LvModule::SetInt()` function. To get the value, use the `LvModule::GetInt32()` or `LvModule::GetInt64()` functions according the type of the passed variable. For convenience Simplon provides also the `LvModule::GetInt()` function, which is equal to `LvModule::GetInt64()`. For example:

```
int32_t BinHor = 1;
if (pDevice->IsReadable(LvDevice_BinningHorizontal))
{
    pDevice->GetInt32(LvDevice_BinningHorizontal, &BinHor);
}

if (pDevice->IsWritable(LvDevice_BinningHorizontal))
{
    pDevice->SetInt(LvDevice_BinningHorizontal, 2);
}
```

The integer value can have defined minimum, maximum and increment. You can retrieve these parameters by the `LvModule::GetInt32Range()` and `LvModule::GetIntRange()` (equal to `LvModule::GetInt64Range()`). The following example shows a function, which adjusts a feature value to the nearest lower possible:

```
int64_t AdjustToValidIntValue(LvDevice* pDevice, LvFeature FeatureID,
                              int64_t Value)
{
    int64_t Min;
    int64_t Max;
    int64_t Inc;
    pDevice->GetIntRange(FeatureID, &Min, &Max, &Inc);
    if (Value < Min) Value = Min;
    if (Value > Max) Value = Max;
    Value = Value - (Value - Min) % Inc;
    return Value;
}
```

2.4.6. The Boolean feature

A Boolean feature can have only 2 values: `true` or `false`. Use the `LvModule::SetBool()` and `LvModule::GetBool()` functions. For example:

```
pDevice->SetBool(LvDevice_ColorTransformationEnable, true);
```

In the Plain C API no type for the Boolean is defined, so the value is of `int32_t` type and has the values of `1` or `0`.

2.4.7. The Float feature

For the floating number representation, the type `double` is used. Use the `LvModule::SetFloat()` and `LvModule::GetFloat()` functions.

To obtain the range of values, use the `LvModule::GetFloatRange()` function. Note that the Increment value is `0` if it is not defined (was not available in the GenICam standard 1.x).

2.4.8. The String feature

The string used in the string feature is an ANSI null-teminated string. Use the `LvModule::SetString()` and `LvModule::GetString()`. Example:

```
char szOldUserID[128];
char szNewUserID[128];
strcpy(szNewUserID, "Chicuelo");

pDevice->GetString(LvDevice_DeviceUserID, szOldUserID,
```

```
                    sizeof(szOldUserID));
pDevice->SetString(LvDevice_DeviceUserID, szNewUserID);
```

In case you need to know in advance how big string buffer you need for the string, use the `LvModule::GetStringSize()`. For example:

```
size_t Size;
pDevice->GetStringSize(LvDevice_EventLvLogMessage, &Size);
char* pLogMessage = new char[Size];
pDevice->GetString(LvDevice_EventLvLogMessage,
                   pLogMessage, Size);
//...
delete[] pLogMessage;
```

The C++ API adds a possibility to use the `std::string` for convenience — see the Section 2.2.2, “The C++ Class Library” [p. 31] for more details. In this case you do not need to take care about the string size. Example:

```
std::string sOldUserID;
std::string sNewUserID = "Tomatito";

pDevice->GetString(LvDevice_DeviceUserID, sOldUserID);
pDevice->SetString(LvDevice_DeviceUserID, sNewUserID.c_str());
```

2.4.9. The Enumeration feature

The enumeration feature can have one value from a list of available values (enumeration entries). In GenICam the enumeration values are *strings*. Simplon for convenience offers also using predefined constants for enumeration entries (included are constants for all enumerations supported by Leutron Vision hardware). Use the `LvModule::SetEnumStr()` and `LvModule::GetEnumStr()` for a **generic** manipulation (enum values as strings) or `LvModule::SetEnum()` and `LvModule::GetEnum()` when you use the **predefined constants**. The following examples show the difference.

Fully generic access:

```
LvFeature FeatureID;
pDevice->GetFeatureByName(LvFtrGroup_DeviceRemote,
                          "PixelFormat", &FeatureID);
pDevice->SetEnumStr(FeatureID, "Mono8");
```

Half generic - enum entry as a string:

```
pDevice->SetEnumStr(LvDevice_PixelFormat, "Mono8");
```

Using predefined constants also for enum entries:

```
pDevice->SetEnum(LvDevice_PixelFormat, LvPixelFormat_Mono8);
```

Note that the actual value of the predefined constant does not have any numerical meaning. An exception is the `LvPixelFormat` — there the constants are compound from several parts, from which is possible to determine additional info, like *bits per pixel*, *mono/color* etc. More info in the Reference Guide.

Sometimes you may need to convert the enum string value to a numeric constant or vice versa. The `LvModule::GetEnumValByStr()` and `LvModule::GetEnumStrByVal()` are available for this purpose.

Similarly as features themselves, some of the enumeration entries may become unavailable upon the current status. Use the `LvModule::IsAvailableEnumEntry()` for determining, if the enum entry is currently available. Typically, the device supports only a subset of all possible pixel formats:

```
if (pDevice->IsAvailableEnumEntry(LvDevice_PixelFormat,
                                  LvPixelFormat_BGR8Packed))
```

```
{
    pDevice->SetEnum(LvDevice_PixelFormat, LvPixelFormat_BGR8Packed);
}
```

More information about the enumeration entries can be obtained using the `LvModule::GetInfo()` and `LvModule::GetInfoStr()`, see Section 2.4.13, “Getting feature properties” [p. 56].

2.4.10. The Command feature

The Command feature enables to execute a command and wait for its completion. To determine, if it is possible to execute the command, check the feature, if it *is writable*, using the `LvModule::IsWritable()` function. To execute the command call the `LvModule::CmdExecute()` function. The command execution can go to a different thread or can be executed outside the application (for example is converted to a GEV message to a camera), in such cases you cannot assume that when the `LvModule::CmdExecute()` returns, the execution of the command is already finished. To determine, whether the command was really completed, use the `LvModule::CmdIsDone()` function.

In case the command execution can influence some other features (see the example below), it is important to call `LvModule::CmdIsDone()` until it returns true. When the command returns true, it internally assures the modified features get the correct values; it also includes a callback on modified features (explained in Advanced topics). Simplon calls `LvModule::CmdIsDone()` internally in case you specify a non-zero timeout as a `LvModule::CmdExecute()` parameter — this can free you from the necessity to write the wait-for-is-done loop.

In the following example is executed the calibration of the camera, which is a quite lengthy process; the waiting here is for simplicity spent in the `Sleep()` function:

```
if (pDevice->IsWritable(LvDevice_LvLensControlCalibrate))
{
    pDevice->CmdExecute(LvDevice_LvLensControlCalibrate);
    bool IsDone = false;
    while (!IsDone)
    {
        pDevice->CmdIsDone(LvDevice_LvLensControlCalibrate,
                           &IsDone);
        if (IsDone) break;
        Sleep(100);
    }
}
```

Note that the call of `LvModule::CmdIsDone()` must include at least one call, which returns `IsDone=true`. If this is not done, the measured values by the calibration will not appear in the corresponding `LvLensControlPlusEnd` and `LvLensControlMinusEnd` features, as these features are *cached* and thus their values must be invalidated after the command is completed — which is what `LvModule::CmdIsDone()` is doing.

2.4.11. The Pointer feature

Use the `LvModule:SetPtr()` and `LvModule::GetPtr()` for this feature. The pointer is an address in operating memory; while this is a natural part of many languages, including C and C++, the languages using MS .Net Framework cannot work with pointers directly due to safety reasons. This issue is discussed in Advanced topics.

Example (get pointer to image data in the LvBuffer):

```
void* pData;
pBuffer->GetPtr (LvBuffer_Base, &pData);
```

2.4.12. The Buffer feature

Use the `LvModule::SetBuffer()`, `LvModule::GetBuffer()` and `LvModule::GetBufferSize()`. This feature works with a pointer (memory address), so in the managed code in the MS .Net Framework cannot be used directly. It is discussed in Advanced topics.

2.4.13. Getting feature properties

A Feature has a number of properties, many of which are useful namely when building a GUI — for example the *Display name*, *Tooltip*, *Description* etc.

These properties can be obtained using the `LvModule::GetInfo()` and `LvModule::GetInfoStr()` functions. The type of the info is defined in the `LvFtrInfo` enumeration. Here are the most common types of info for a feature:

- `LvFtrInfo_IsStreamable` - Returns 1 if the feature has the Streamable attribute set. To be used in the `LvModule::GetInfo()` function.
- `LvFtrInfo_Name` - Returns the feature Name. Do not confuse it with the DisplayName - the Name is the string identifier, by which the feature can be identified and a numeric ID can be obtained for further actions (generic feature access). To be used in the `LvModule::GetInfoStr()` function.
- `LvFtrInfo_DisplayName` - Returns the feature Display name for representation in GUI. To be used in the `LvModule::GetInfoStr()` function.
- `LvFtrInfo_Description` - Returns the feature Description text. To be used in the `LvModule::GetInfoStr()` function.
- `LvFtrInfo_PhysicalUnits` - Returns the feature Physical units, if defined. To be used in the `LvModule::GetInfoStr()` function.
- `LvFtrInfo_ToolTip` - Returns the feature Tooltip (a short description to be used in the GUI). To be used in the `LvModule::GetInfoStr()` function.

Similarly as for features, an info can be obtained for enumeration entries of the enumeration features. The `Param` in the `LvModule::GetInfo()` specifies a zero based index of the entry or the Simplon enum entry constant. You can obtain the number of entries by the `LvModule::GetInfo()` function with the `LvFtrInfo_EnumEntryCount` parameter. If the `Param` is set to `LV_ENUMENTRY_CURRENT`, the returned info is for the currently selected enum entry. Here are some of them:

- `LvFtrInfo_EnumEntryName` - Returns the symbolic name of the enum entry. To be used in the `LvModule::GetInfoStr()` function.
- `LvFtrInfo_EnumEntryDisplayName` - Returns the display name of the enum entry. To be used in the `LvModule::GetInfoStr()` function.
- `LvFtrInfo_EnumEntryDescription` - Returns the description of the enum entry. To be used in the `LvModule::GetInfoStr()` function.
- `LvFtrInfo_EnumEntryToolTip` - Returns the tool tip of the enum entry. To be used in the `LvModule::GetInfoStr()` function.

- `LvFtrInfo_EnumEntryAccess` - Returns the access of the enum entry (one of the `LvFtrAccess` constants). To be used in the `LvModule::GetInfo()` function.
- `LvFtrInfo_EnumEntryCount` - Returns the number of enum entries for the enum. To be used in the `LvModule::GetInfo()` function.
- `LvFtrInfo_EnumEntryNameMaxSize` - Returns the maximum string size needed (including terminating zero) for any entry name of the enum. To be used in the `LvModule::GetInfo()` function.

2.4.14. Saving and loading a configuration

A common task is to save the device configuration so that it can be restored next application run. The principle of saving the configuration is saving all features, which have the *Streamable* flag set. This flag expresses the fact, that the feature is a part of the device configuration. The saving of features is trivial, however, restoring the features is a bit tricky: for example the `OffsetX` feature is not writable until the `Width` feature is not set to a value lower than is the maximum width. That means the order of the features set during the restore process is significant. For this reason Simplon offers functions for saving and restoring Device features: `LvDevice::SaveSettings()` and `LvDevice::LoadSettings()`.

2.4.15. Building a feature tree

In case you would need to create a tree of features, like it is in Simplon Explorer, you need to obtain the information about the feature *level*. The level is returned in the last parameter of the `LvModule::GetFeatureAt()` function. The following piece of code prints all features of the Device, indented according to the level:

```
uint32_t NumFeatures;
pDevice->GetNumFeatures (LvFtrGroup_DeviceRemote, &NumFeatures);
for (uint32_t Index = 0; Index < NumFeatures; Index++)
{
    LvFeature Feature;
    uint32_t Level;
    std::string sName;
    pDevice->GetFeatureAt (LvFtrGroup_DeviceRemote,
                           Index, &Feature, &Level);
    pDevice->GetInfoStr (Feature, LvFtrInfo_Name, sName);
    char szIndent[128];
    memset(szIndent, ' ', sizeof(szIndent));
    szIndent[Level*3] = 0;
    printf("%s %s", szIndent, sName.c_str());
}
```

2.5. Writing maintainable applications

The GenICam standard requires that the IDs of the modules, like is the Interface and Device module, are to be unique. For example 2 cameras of exactly the same type and configuration will still have different IDs (to assure this, the ID is usually derived from a unique feature, like is the MAC address or serial number).

In the source code the IDs are required for opening the modules, but it might not be a good idea to hardcode them directly — in such case the application would stop working, when you for example replace a defective camera (= Device) by another piece (even of the same type) or change the NIC card (= Interface). In GUI applications you can enumerate IDs of available interfaces and devices and offer them to the user, which then selects the desired one. But this is not the typical case in the industry, where the application usually should work automatically. The application should be maintainable, that means when there is a need to change the hardware (because of an update or repair), there must be an easy way, how to make the application working with the changed hardware.

The simplest solution is to place the IDs to a configuration file, so that after the HW change the new IDs can be changed in the configuration file. However, this still requires a skilled service person.

Let's have a look how to write applications, which in optimal case do not require any change, if the hardware is changed. Unfortunately, there is no one universal solution; the solution differs from case to case. But its principle is equal: instead of direct hardcoding the IDs in the source code, or storing them, rather *make a search* for them.

2.5.1. Maintainable Interface ID

The situation with the Interface is usually not complex to solve.

The following code shows how to iterate through all available interfaces and check some of its features, for example its IP address (no error handling included):

```
bool FindInterfaceByIpAddress(LvSystem* pSystem,
                              std::string sIpAddress,
                              std::string& sInterfaceId)
{
    pSystem->UpdateInterfaceList();
    int32_t MinIndex, MaxIndex, Increment;
    pSystem->GetInt32Range(LvSystem_InterfaceSelector,
                           &MinIndex, &MaxIndex, &Increment);
    for (int32_t i = MinIndex; i <= MaxIndex; i += Increment)
    {
        pSystem->SetInt(LvSystem_InterfaceSelector, i);
        std::string sFoundIpAddress;
        pSystem->GetString(LvSystem_GevInterfaceDefaultIPAddress,
                           sFoundIpAddress);
        if (strstr(sFoundIpAddress.c_str(), sIpAddress.c_str())!=NULL)
        {
            pSystem->GetString(LvSystem_InterfaceID,
                               sInterfaceId);
            return true;
        }
    }
    sInterfaceId = "";
    return false;
}
```

Simplon offers a native function, which does exactly the same: `LvSystem::FindInterface()`. If you have a look at this function in the Reference Guide, you will see that the search criteria is set by the `FindBy` parameter. In case of Interface, this can be `LvFindBy_TLType`, `LvFindBy_DisplayName` or `LvFindBy_GevIPAddress`. The function code above could be then replaced by one line:

```
bool FindInterfaceByIpAddress(LvSystem* pSystem,
                              std::string sIpAddress,
                              std::string& sInterfaceId)
{
    return pSystem->FindInterface(LvFindBy_GevIPAddress,
                                  sIpAddress.c_str(),
                                  sInterfaceId);
}
```

In case you have always only one interface of given type in the system (only one frame grabber present in the system, or only one NIC card), the most useful way is to search for the interface *by its type*, using the `LvFindBy_TLType`. The following code searches for the first interface of the GEV (GigE Vision) type.

```
if (pSystem->FindInterface(LvFindBy_TLType, "GEV",
                           sInterfaceId))
{
```

```
    //...
}
```

Note that Simplon by default offers **all** GEV devices on a single interface, regardless of number of NIC involved, thus finding the GEV interface by TLType is recommended also for GigE devices.

2.5.2. Maintainable Device ID

The following code snippet implements searching for a device according to the model name. The returned device ID can be then used for opening the device. For simplicity, the error handling is not included in this code.

```
bool FindDeviceByModelName(LvInterface* pInterface,
                           std::string sModelName,
                           std::string& sDeviceId)
{
    pInterface->UpdateDeviceList();
    int32_t MinIndex, MaxIndex, Increment;
    pInterface->GetInt32Range(LvInterface_DeviceSelector,
                              &MinIndex, &MaxIndex, &Increment);
    for (int32_t i = MinIndex; i <= MaxIndex; i += Increment)
    {
        pInterface->SetInt(LvInterface_DeviceSelector, i);
        std::string sFoundModelName;
        pInterface->GetString(LvInterface_DeviceModelName,
                              sFoundModelName);
        if (strstr(sFoundModelName.c_str(), sModelName.c_str())!=NULL)
        {
            pInterface->GetString(LvInterface_DeviceID,
                                  sDeviceId);
            return true;
        }
    }
    sDeviceId = "";
    return false;
}
```

Note that when you replace the `LvInterface_DeviceModelName` constant, principally the same code can be used for searching the device according its IP address (`LvInterface_GevDeviceIPAddress`), MAC address (`LvInterface_GevDeviceMACAddress`), vendor name (`LvInterface_DeviceVendorName`) or the User ID (`LvInterface_DeviceUserID`).

The Simplon native function `LvInterface::FindDevice()` provides the same functionality:

```
bool FindDeviceByModelName(LvInterface* pInterface,
                           std::string sModelName,
                           std::string& sDeviceId)
{
    return pInterface->FindDevice(LvFindBy_ModelName,
                                  sModelName.c_str(), sDeviceId);
}
```

The selection of search criteria depends on an actual hardware configuration.

- If there are always devices of different types (for example only 2 cameras used, each of a different type), the search according the model type is suitable (use `LvFindBy_ModelName` in `LvInterface::FindDevice()`). In such case there is no reconfiguration needed when you change a defective camera by another piece of the same type.
- If the cameras are of the same type and are of the GigE type and fixed IP addresses are used, you can obtain the camera ID upon the IP address (use `LvFindBy_GevIPAddress`). In case you exchange a defective camera, the new one must be assigned the same IP address or its actual IP address must be stored in some configuration file.

- If the cameras are of the type and a User ID (nickname) can be assigned to each, you can obtain the ID upon the User ID (use `LvFindBy_UserID`). In case a defective camera is to be replaced, the new one has to be configured to the same User ID, as previous or its actual User ID must be stored in some configuration file.
- When using the GigE cameras, you can never rely on the camera order, as it is usually random and depends on the order of discovery of the cameras in the network. However, if the cameras are on a frame grabber, they are reported always in the same order, so you can simply use the camera *Index* as an identifier:

```
pInterface->GetDeviceId(Index, sDevice);
```

2.6. Deploying your application

If you want to distribute your application written with Simplon, the Simplon libraries must be installed by the Setup, it is not enough just to copy the libraries. In case you want to embed the Simplon setup in your setup, or configure the setup for automatic run, please read the Simplon getting started chapter, where is described how to customize Simplon setup.

2.6.1. Windows

Simplon Setup adds its BIN folder to the PATH environment variable, so your application does not need to reside in the same folder as Simplon executable files.

2.6.2. Linux

Simplon installer configures the dynamic linker, adding links to Simplon API libraries, so they will be properly located and loaded at runtime.

3. Advanced topics

3.1. Saving images to files

Simplon offers a possibility to save the image to a file in 3 common bitmap file formats: BMP, TIFF and JPEG. If the pixel format of the image is not supported by the file format, the image is automatically converted to acceptable pixel format before saving, for example 12–bit monochrome pixel format is converted to 8–bit monochrome for BMP and JPEG and and may be converted to 16–bit monochrome for TIFF. The functions for saving are belonging to the Buffer module.

The **BMP** is the simple Windows Bitmap format, which is suitable for 8–bit monochrome, or 24 and 32–bit BGR formats. The bitmap is reverted to bottom-up line order (the top-down order is not supported by many software packages). There is no compression used.

```
pBuffer->SaveImageToBmpFile("C:\Data\Images\SampleImage.bmp");
```

The **TIFF** format is another uncompressed format, which can save 8–bit, 10–bit, 12–bit and 16–bit monochrome, 24 and 32–bit RGB formats. It is provided namely for the possibility to save images in 10 or 12–bit monochrome format. In the `Options` parameter you can use the `LvipOption_TiffConvertTo16Bit` flag which automatically converts the image to 16–bit monochrome pixel format — this can be helpful, because many software packages do not treat 10 and 12–bit formats in TIFF properly.

```
pBuffer->LvBufferSaveImageToTifFile("C:\Data\Images\SampleImage.tif",
                                    LvipOption_TiffConvertTo16Bit);
```

The **JPEG** format is a common format for photographic images, with a lossy compression of selectable compression factor. It natively supports only 8–bit monochrome or 24–bit BGR formats. The compression is set by the quality factor in the range from 0 to 100 (the higher the quality factor, the lower the compression). In the sample below we use the factor 95.

```
pBuffer->LvBufferSaveImageToJpgFile("C:\Data\Images\SampleImage.jpg",
                                    95);
```

The Simplon API enables only to *write* the images to files. If you want to *read* them, you can utilize the *Image preprocessing library* (see Chapter 5, *The image preprocessing library* [p. 85]) — in fact Simplon uses it as well for saving.

3.2. White Balance, Gamma, Contrast, Brightness

The image from a color camera usually needs a *white balance* correction. Simple white balance implementation means that each of the R,G and B pixel values is multiplied by a correction factor for this color. For example if the image is too red, the R value of all pixels should be reduced by a calculated factor. The factors can be easily calculated when the camera is aimed to a neutral grey object. Usually the factors are calculated so that all are greater than or equal to 1. This assures that white remains white in the image, but it also increases a bit the overall brightness.

The *brightness* and *contrast* are simple linear operations; the brightness adds a constant to a pixel value, the contrast uses a multiplication operation to lighten the light pixels and darken the dark pixels.

The gamma correction is a non-linear adjustment of the image pixel values. While the darkest and lightest pixels are remaining almost unchanged, the value of medium value pixels can be increased (lightening the medium tones) or decreased (darkening the medium tones). The gamma correction requires a floating point operation for each pixel.

Although the application of the operations described above is quite simple, these operations can significantly load the CPU when applied to every pixel of a large image. Thus, it is more efficient to precalculate the required operations for each pixel value and use a precalculation table, so

called lookup table (LUT), the application of which means only one redirection operation for each pixel, so it is much faster.

In Simplon Explorer, when you click on "Show Image Processing Dialog", the dialog with the graphically represented LUT is displayed, so you can get an overview, how the LUT is changing, when you modify some of the above mentioned parameters. In case of color RGB cameras, the LUT is composed from parts, one for each color channel.

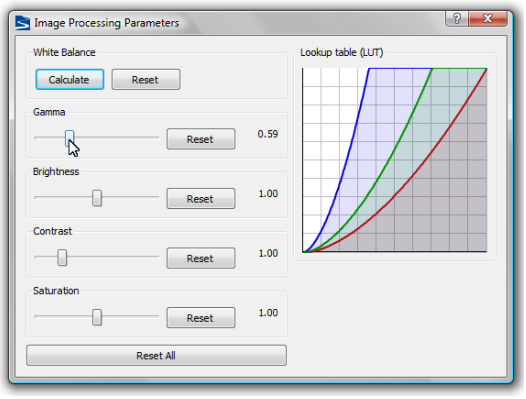


Figure 3.1. Simplon Explorer — Image Processing Parameters

The LUT can be either directly on the device (hardware LUT) or implemented by software on the receiver side. Both LUTs are available as device features; the HW LUT is in the Device Remote Features, the SW LUT is in the Simplon Device Features. Furthermore, the Simplon Device Features offer the generation of LUT from white balance, gamma, brightness and contrast (the `LvUniLUTMode` mode must be `Generated`), and automatically set the HW LUT, if it is available. More info in the chapter Section 3.3, “Unified image preprocessing” [p. 62].

3.3. Unified image preprocessing

The cameras may differ in capabilities of image preprocessing; while some model can do the Bayer color decoding and applying LUT directly in hardware (such models are marked as *RTF* in our hardware products), other models may not be capable of such preprocessing. If your application is expected to work with various camera models, you can write the application so that it checks the capabilities of the camera and either utilize the hardware processing, or do the processing by software in your application, using for example the Simplon *Image preprocessing library* (see Chapter 5, *The image preprocessing library* [p. 85]).

However, to handle such situations properly requires quite a complex code. For this reason Simplon offers a *unified image preprocessing*, the idea of which is that the preprocessing is handled automatically and your code is simple and universal for all camera models. The automatic handling means that Simplon determines the camera capabilities and automatically substitutes the image processing by software if it is not provided by the hardware. You can check how it works in the Simplon Explorer — it also utilizes this functionality, when you have the *Automatic image processing* switched on:

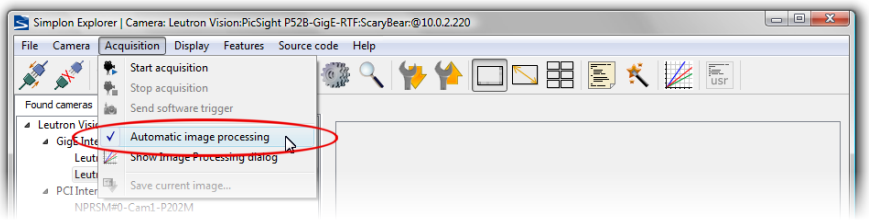


Figure 3.2. Simplon Explorer — Automatic image processing

The unified image preprocessing uses a chain of processing functions, which includes *Bayer Decoding* - *LUT* (brightness, contrast, gamma, white balance) - *Color Correction* (saturation) - *Pixel Format Conversion*. The software processing is done immediately after the image is picked from the output buffer queue, just before it is passed to your application.

The processing chain is determined automatically; in case it does not fit to your needs and you need some more specific image processing, you can always switch off the automatic image processing and use the Image Processing Library directly instead.

3.3.1. Source code adaptation

To start using the unified image preprocessing needs only a few changes in the source code. First, you should set the processing mode to Auto:

```
pDevice->SetEnum(LvDevice_LvUniProcessMode, LvUniProcessMode_Auto);
```

Note that setting the `LvUniProcessMode` to `LvUniProcessMode_HwOnly` means switching off the automatic software processing.

Then set the desired target pixel format by the `LvDevice_LvUniPixelFormat` feature, for example:

```
if (CameraIsColor)
    pDevice->SetEnum(LvDevice_LvUniPixelFormat,
                     LvPixelFormat_BGRA8Packed);
else
    pDevice->SetEnum(LvDevice_LvUniPixelFormat,
                     LvPixelFormat_Mono8);
```

This determines the processing chain, for example if the camera is Bayer array, the Bayer decoding is automatically included in the chain. Other processing functions are added to the chain, when you set further parameters, like *gamma*, *white balance*, *saturation* etc.

Then, if you want to get the pointer to the image data, instead of `LvBuffer_Base` feature use the `LvBuffer_UniBase` feature. This feature returns a pointer either to the original image if SW processing was not needed, or to the processed buffer if the SW processing was done.

```
void* pData = NULL;
pBuffer->GetPtr (LvBuffer_UniBase, &pData);
```

When you work with `LvBuffer_UniBase` and want to get the image parameters, use the `LvDevice_LvUniPixelFormat`, not the `LvDevice_PixelFormat`. This assures you work always with a correct pixel format — if there is a difference between the source and target pixel formats, the SW processing will always take place, so the `LvDevice_LvUniPixelFormat` is always the right one of the image to which you obtain the pointer by `LvBuffer_UniBase`.

```
pDevice->GetEnum(LvDevice_LvUniPixelFormat, &PixelFormat);
```

For convenience, also the line pitch (increment in bytes) is supplied, in the `LvDevice_LvUniLinePitch` feature. The image Width and Height are not changed, so you can use the `LvDevice_Width` and `LvDevice_Height` features.

Note that the *Project generation wizard* in Simplon Explorer offers an option *Include automatic image processing* by many templates; this option adapts the source code in the similar way, as is explained above.

3.3.2. Preprocessing parameters

The preprocessing parameters can be set by the Device features under the *Unified Processing* category in the feature tree.

3.3.2.1. Processing mode (LvUniProcessMode)

- **Hardware Only**: Hardware only processing — the requested processing is done only if it is available on the hardware. If it is missing, it is not substituted by software processing. This is the default setting and in fact it means the automatic image preprocessing is switched off.
- **Software Only**: Software only processing — even if the requested processing is available in hardware, it is substituted by the software processing.
- **Auto**: Automatic mode: If the requested processing can be done by the hardware, it is done in hardware, otherwise it is substituted by software processing.

3.3.2.2. Enable in-place processing (LvUniProcessEnableInPlace)

This boolean feature indicates, whether the processing can be in-place, that means the result of processing is stored to the original image buffer and not to the process buffer. Even if this feature is set to true, depending on the actual processing chain, it may not be possible to use the in-place processing, for example in case the target pixel format is different or the processing algorithm needs to work with multiple source pixels.

3.3.2.3. Unified pixel format (LvUniPixelFormat)

This feature specifies the target pixel format of the processed image. If the source image has different pixel format, the image is converted to this pixel format in the processing chain. If the source pixel format is Bayer array format, the software Bayer decoding is used for the conversion to the target pixel format (if you want to use the hardware Bayer decoding, set the *source* pixel format to one of the BGR or RGB pixel formats).

3.3.2.4. Bayer decoding algorithm (LvUniBayerDecoderAlgorithm)

There are several algorithms available in the software Bayer decoding . They are ordered by the speed: the higher the speed, the lower the result quality.

- **Nearest Neighbour**: The fastest method for Bayer array decoding. It uses the nearest pixel with the required lens color to get the pixel value. Gives rough results.
- **Bilinear Interpolation**: The most commonly used method for fast Bayer decoding. For the color not directly available for the given pixel makes the linear interpolation between the 2 or 4 neighbouring pixels to get it. Gives good results with a high speed.
- **Bilinear Color Correction**: The interpolation with Linear Color Correction (LCC) is another adaptive algorithm and optimized for images with edges in horizontal and vertical direction.
- **Pixel Grouping**: A method similar to the *Variable Gradient*, but simplified and thus faster, still giving very good results.
- **Variable Gradients**: One of the best known methods for Bayer decoding, but significantly slower than the bilinear interpolation. It is based on evaluation the color gradients in 8 directions around the pixel and selecting the set of best set for the color interpolation. The slowest algorithm.

If the Bayer decoding is used in hardware, this feature may be ignored, as the hardware usually provides only one Bayer decoding method.

3.3.2.5. LUT control

Similarly as Bayer decoding, the LUT may be available on the hardware or is applied in software. You can either set the LUT directly from your application (in this case the `LvDevice_LvUniLUTMode` must be set to `LvUniLUTMode_Direct`), *or* let the LUT to be calculated from *white balance*, *gamma*, *brightness* and *contrast* (in this case the `LvDevice_LvUniLUTMode` must be set to `LvUniLUTMode_Generated`).

The following sample code shows how to directly an inverse LUT for monochrome pixel format:

```
pDevice->SetEnum(LvDevice_LvUniLUTMode, LvUniLUTMode_Direct);
pDevice->SetEnum(LvDevice_LvUniLUTSelector, LvUniLUTSelector_Luminance);
```

```
int32_t MaxIndex, MaxValue;
pDevice->GetInt32Range(LvDevice_LvUniLUTIndex, NULL, &MaxIndex, NULL);
pDevice->GetInt32Range(LvDevice_LvUniLUTValue, NULL, &MaxValue, NULL);
for (int i=0; i<=MaxIndex; i++)
{
    pDevice->SetInt(LvDevice_LvUniLUTIndex, i);
    pDevice->SetInt(LvDevice_LvUniLUTValue, MaxValue-i);
}
```

However, the code above is slow, namely when the hardware LUT is affected. For the case you need to modify the whole LUT it is better to use an alternative method for the transfer of the whole LUT at once:

```
pDevice->SetEnum(LvDevice_LvUniLUTMode, LvUniLUTMode_Direct);
pDevice->SetEnum(LvDevice_LvUniLUTSelector, LvUniLUTSelector_Luminance);
uint32_t Lut[4096];
for (uint32_t i=0; i<4096; i++)
    Lut[i] = 4095-i;
pDevice->SetBuffer(LvDevice_LvUniLUTValueAll, Lut, sizeof(Lut));
```

Note that in the code above, the Luminance LUT is 12–bit, thus it has 4096 values. In case of RGB LUT there are 256 values for each LUT, so the RGB LUT is 8–bit. However, in both cases the LUT values are stored in the 32–bit values and in both cases the maximum reported LUT size is 4096 x 4 = 16384 bytes. The same applies to the Remote Device LUT (the HW LUT), but depending on the hardware type, the LUT values migt be stored either in Big Endian or Little Endian. This applies to Leutron devices; on šrd party devices the LutValueAll may work with a different buffer layout.

To make the work with the LUT more comfortable, Simplon also offers an API for UniLUT. The LUT passed as the parameter can be 8–bit, 10–bit or 12–bit.

```
uint8_t LUT[256];
for (int i=0; i<256; i++)
    LUT[i] = 255-i;
pDevice->UniSetLut(LvLUTSelector_Luminance, LUT, sizeof(LUT));
```

The following sample code shows how to set the *white balance factors*, *gamma*, *brightness* and *contrast*. As a result of all these parameters is a calculated LUT, the application of which is much faster than performing the calculation on each pixel.

```
pDevice->SetEnum(LvDevice_LvUniLUTMode, LvUniLUTMode_Generated);
pDevice->SetEnum (LvDevice_LvUniBalanceRatioSelector,
                  LvUniBalanceRatioSelector_Red);
pDevice->SetFloat(LvDevice_LvUniBalanceRatio, 1.0);

pDevice->SetEnum (LvDevice_LvUniBalanceRatioSelector,
                  LvUniBalanceRatioSelector_Green);
pDevice->SetFloat(LvDevice_LvUniBalanceRatio, 1.28);

pDevice->SetEnum (LvDevice_LvUniBalanceRatioSelector,
                  LvUniBalanceRatioSelector_Blue);
pDevice->SetFloat(LvDevice_LvUniBalanceRatio, 1.35);

pDevice->SetFloat(LvDevice_LvUniGamma, 1.2);
pDevice->SetFloat(LvDevice_LvUniBrightness, 0.9);
pDevice->SetFloat(LvDevice_LvUniContrast, 1.1);
```

Whenever you set one of these features, the LUT is recalculated from scratch, that means you cannot combine these features with direct LUT settings; one excludes the other.

The **white balance** factors can be calculated from the image:

```
pDevice->SetEnum(LvDevice_LvUniBalanceWhiteAuto,
                 LvUniBalanceWhiteAuto_Once);
```

If there is already a buffer with a grabbed image available (and the buffer is not yet returned to the input buffer pool), the calculation is done from this buffer. Otherwise setting this enum only

prepares for the calculation of the factors; the calculation itself is done when the next image is acquired and this enum is reset. Alternatively, you can call the factors calculation on a Buffer:

```
pBuffer->UniCalculateWhiteBalance();
```

This calculation is done immediately, because it is known from which buffer the factors are to be calculated.

3.3.2.6. Color transformation control

The color transformation applies a 3x3 matrix to the R,G and B channels of an RGB pixel format. Similarly as with the LUT, the matrix can be set directly:

```
pDevice->SetBool(LvDevice_LvUniColorTransformationEnable, true);

pDevice->SetEnum (LvDevice_LvUniColorTransformationValueSelector,
                  LvUniColorTransformationValueSelector_Gain00);
pDevice->SetFloat(LvDevice_LvUniColorTransformationValue, 0.8);

pDevice->SetEnum (LvDevice_LvUniColorTransformationValueSelector,
                  LvUniColorTransformationValueSelector_Gain01);
pDevice->SetFloat(LvDevice_LvUniColorTransformationValue, 0.12);

//...

pDevice->SetEnum (LvDevice_LvUniColorTransformationValueSelector,
                  LvUniColorTransformationValueSelector_Gain22);
pDevice->SetFloat(LvDevice_LvUniColorTransformationValue, 0.98);
```

Or you can let the matrix to be calculated from the *saturation* factor:

```
pDevice->SetFloat(LvDevice_LvUniSaturation, 0.91);
```

3.3.3. Additional buffers for processing

The image preprocessing usually requires to allocate for each image buffer an additional buffer, to which is stored the processed image. You can let Simplon to allocate such buffer automatically, or supply your own buffers.

3.3.3.1. Allocation of process buffers by the application

The following code snippet illustrates how to allocate own buffers and with the `LvBuffer::AttachProcessBuffer()` function add processing buffers. The needed size of the processing buffer is obtained form the `LvUniProcessPayloadSize` feature of the Device. Error handling is omitted for simplicity.

```
LvBuffer* Buffers[NUMBER_OF_BUFFERS];
int32_t BufSize;
pStream->GetInt32(LvStream_LvCalcPayloadSize, &BufSize);
int32_t ProcessBufSize;
pDevice->GetInt32(LvDevice_LvUniProcessPayloadSize, &ProcessBufSize);
for (int i=0; i<NUMBER_OF_BUFFERS; i++)
{
    void* pData = malloc(BufSize);
    pStream->OpenBuffer(pData, BufSize, NULL, 0, Buffers[i]);
    void* pProcessData = malloc(ProcessBufSize);
    Buffers[i]->AttachProcessBuffer(pProcessData, ProcessBufSize);
}
```

It important to know, that the value of this feature is dependent on the features `Width`, `Height`) and `LvUniPixelFormat`, so get the `LvUniProcessPayloadSize` only after you completely configure these features.

3.3.3.2. Automatic process buffer allocation

If your application does not supply own process buffers, the buffers are allocated automatically. The allocation is postponed till the time the buffer is really needed. Under certain circumstances the processing can be omitted, for example the application needs to set *gamma* using a LUT. However, when the gamma is 1.0, and applying the resulting LUT would give the same image data. In such case the processing is omitted and directly the source image is used. However, when gamma changes to a different value, the processing must be done. If at this moment is detected that the process buffer was not yet allocated, it is allocated automatically. If it is detected that the allocated buffer does not have enough size, it is reallocated. So it is important to keep in mind that your application should never remember the pointer from the `LvBuffer_UniBase` feature, as it can change next time an image is acquired; instead it should always ask for it by the `LvBuffer::GetPtr()` function.

If you for some reason want to allocate all process buffers at once, you can call the `LvPreallocateProcessBuffers` command feature.

```
pStream->CmdExecute(LvStream_LvPreallocateProcessBuffers);
```

The automatic allocation can be disabled by the `LvAutoAllocateProcessBuffers` feature (by default this feature is true). Use this if you want to be sure no automatic allocation can happen.

```
pStream->SetBool(LvStream_LvAutoAllocateProcessBuffers, false);
```

3.4. Processing chunk data

Chunk data are data appended to the image data, containing additional image information, like a timestamp, frame ID, exposure etc. You can see that the chunk data appear not as *Buffer* features, but rather as *Device* features — see the feature tree, the items in the *Chunk data control* category. These features are not automatically updated after each acquired image; to do so, you must add to your code the call of the `LvBuffer::ParseChunkData()` function.

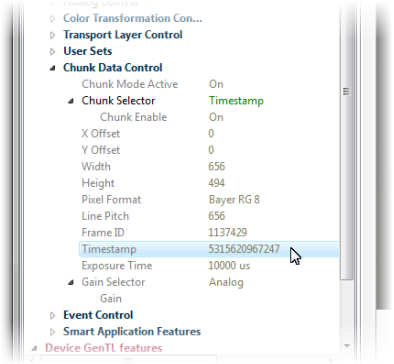


Figure 3.3. Simplon Explorer — Chunk data

Suppose we want to get the *Timestamp* and the *FrameID* chunk data. Before starting the acquisition, these features must be enabled:

```
pDevice->SetBool(LvDevice_ChunkModeActive, true);

pDevice->SetEnum (LvDevice_ChunkSelector, LvChunkSelector_FrameID);
pDevice->SetBool(LvDevice_ChunkEnable, true);

pDevice->SetEnum (LvDevice_ChunkSelector, LvChunkSelector_Timestamp);
pDevice->SetBool(LvDevice_ChunkEnable, true);
```

In the new image handler (the image callback function or after the `LvEvent::WaitAndGetData()` function call) call the `LvBuffer::ParseChunkData()`

function. This function parses the chunk data of this *Buffer* to the features of the *Device*. Then you can read the features.

```
void CCamera::CallbackNewBuffer(LvBuffer* pBuffer)
{
    int64_t LvChunkFrameID;
    int64_t ChunkTimeStamp;
    pBuffer->ParseChunkData();
    pDevice->GetInt64(LvDevice_ChunkFrameID, &LvChunkFrameID);
    pDevice->GetInt64(LvDevice_ChunkTimestamp, &ChunkTimeStamp);
    // ...
    pRenderer->DisplayImage(pBuffer);
    pBuffer->Queue();
}
```

3.5. Native functions

Native functions are functions, which enable to set or use some functionality more effectively or faster than through the direct usage of *features*. A typical example is the LUT — setting the contents of 12–bit LUT would mean to 4096 times set the `LvDevice_LUTIndex` and 4096 times set the `LvDevice_LUTValue`. Even if the feature access is optimized for speed, this can take significant amount of CPU power. A native function in such case takes an array of values as a parameter and thus is more comfortable and faster.

3.5.1. Setting the LUT

For setting and getting the LUT use the `LvDevice::UniSetLut()` and `LvDevice::UniGetLut() functions.`

Simplon supports 3 types of LUTs: 8–bit (1 byte per LUT value), 10–bit and 12–bit (2 bytes per LUT value). Thus the LUT for one component can have only one of the following 3 sizes:

- 256 bytes for 256 values of 8–bit LUT
- 2048 bytes for 1024 values of 10–bit LUT
- 8192 bytes for 4096 values of 12–bit LUT

For color pixel formats the LUT consists of 3 components — Red, Green and Blue. The following sample code shows, how to set an inverted LUT for a monochrome pixel format.

```
uint8_t LUT[256];
for (int i=0; i<256; i++)
    LUT[i] = 255-i;
pDevice->UniSetLut(LvLUTSelector_Luminance, LUT, sizeof(LUT));
```

Although the `LvDevice::UniSetLut()` belongs to the Unified preprocessing functions (see Section 3.3, “Unified image preprocessing” [p. 62]), it also enables to set directly the hardware LUT, if you switch off the preprocessing (the `LvUniProcessMode` feature to `HwOnly`) or use the `LvUniLutFlags_HwLut` option.

Simplon takes care about automatic LUT conversion; you can for example use the 12–bit LUT for an 8–bit pixel format or vice versa, because internally the LUT is always synchronized in all 3 formats and an appropriate format is used.

3.6. Simplon log output

Simplon sends log messages to the `lv.simplon.log` file (and optionally to other log message receivers). See the Simplon getting started for more info about the logging.

Note that the log messages are not designed to be always understandable by the end user, the log is intended namely as a tool for the technical support staff to get sufficient info about the system and the application.

3.6.1. Writing to Simplon log

You might need to utilize the logging also for your purposes, or save to the log additional info useful for the technical support. You can send a log message simply by the `LvLog` function, for example:

```
LvLog("Starting the conversion.");
```

It appears in the log with the level=Info and the message is prefixed with `"U: "`.

3.7. Feature callbacks

The feature *value* or *status* can change:

- **On dependence on changing other feature(s)**. For example if the `Width` feature is set to its maximum possible value, the `OffsetX` feature becomes read-only. Or if the acquisition is started, many read-write features become read-only (for example `PixelFormat`) and vice versa.
- **On dependence on a physical status**, for example the `LvDeviceUpTime` or `DeviceTemperature` are changing independently on any user action.
- **On specific event**. Simplon may be notified through a standard GenICam mechanism that a specific event occurred on a remote device, and this event results in new values in some features. A typical example are log messages sent from the PicSight GigE Smart camera.

The application, which uses the features, should be able to react on such changes. An example is Simplon Explorer — there you can see that the feature tree is updated according to the current status. Although your application will probably not display all the features like the Simplon Explorer does, there are some cases when you might need to be able to react on such changes, for example when you want to continuously display a device temperature.

A simple solution is to read a value, which is expected to change, periodically, with an interval corresponding to the nature of the feature. But in some cases such a simple loop with reading the features might not function properly; for example when reading log messages from a camera such approach could cause loss of some messages, when not reading them fast enough.

Simplon offers a possibility to register for each feature a callback function, which is called whenever the feature changes its value or its status.

Let's illustrate the usage of this callback on a simple example. Suppose we have a CCamera class, which represents one camera and contains a pointer to the `LvDevice` class in the `m_pDevice` member variable. We will need to register callbacks for 2 features: `LvDeviceUpTime` and `DeviceTemperature`. Instead of defining a callback function for each feature, we create a helper class `CFeature`, to which we wrap the necessary info and to which we can obtain a pointer in the callback. The `CallbackFeatureUpdated()` function repaints the feature value on the screen (implementation omitted here)

```
class CFeature
{
public:
    LvModule* m_pModule;
    LvFeature m_FeatureId;
    void CallbackFeatureUpdated();
};

void CFeature::CallbackFeatureUpdated()
{
    int32_t Value;
    m_pModule->GetInt32(m_FeatureId, &Value);
```

```
    // now update the value in the GUI
}
```

Then, we will create a callback function, which will be registered in Simplon. This cannot be a member function of a class, it must be a plain C function:

```
void LV_STDC CallbackFeatureUpdatedFn (void* pUserParam,
                                       void* pFeatureParam,
                                       const char* pName)
{
    CCamera* pCamera = (CCamera*) pUserParam;
    pCamera->CallbackFeatureUpdated(pFeatureParam, pName);
}
```

Note that the function receives 3 parameters, the first 2 are parameters which we supply during the registration, so we can put there pointers to `CCamera` and `CFeature` instances. The third one is the feature name, we do not utilize this one in our sample. The `CCamera::CallbackFeatureUpdated()` function is defined as follows.

```
void CCamera::CallbackFeatureUpdated(void* pFeatureParam,
                                     const char* pName)
{
    CFeature* pFeature = (CFeature*) pFeatureParam;
    pFeature->CallbackFeatureUpdated();
}
```

So now you see that we can use a single callback function for multiple features (the advantage of this approach becomes obvious namely when you need to react to dozens of features) and the callback is parsed through `CCamera::CallbackFeatureUpdated()` to `CFeature::CallbackFeatureUpdated()`, where we can react on the particular feature change.

The callback function must be registered for each feature we are interested in. We will work with multiple features, and instead of having a variable for each `CFeature` instance, we use a *list* of these features (for possibility to deallocate at the end), so we utilize the *vector* template from the standard template library:

```
typedef std::vector<CFeature*> CFeatureVector;
```

And in the `CCamera` class we have the vector in a member variable:

```
CFeatureVector m_FeatureVector;
```

The callback registration for each feature is done as follows:

```
void CCamera::RegisterFeatureCallbacks()
{
    CFeature* pFeature;

    pFeature = new CFeature();
    pFeature->m_pModule = m_pDevice;
    pFeature->m_FeatureId = LvDevice_LvDeviceUpTime;
    m_FeatureVector.push_back(pFeature);
    m_pDevice->RegisterFeatureCallback(pFeature->m_FeatureId,
                                       CallbackFeatureUpdatedFn,
                                       this, pFeature);

    pFeature = new CFeature();
    pFeature->m_pModule = m_pDevice;
    pFeature->m_FeatureId = LvDevice_DeviceTemperature;
    m_FeatureVector.push_back(pFeature);
    m_pDevice->RegisterFeatureCallback(pFeature->m_FeatureId,
                                       CallbackFeatureUpdatedFn,
                                       this, pFeature);
}
```

As you can see, we pass *this* and *pFeature* as the third and fourth parameters in the `LvDevice::RegisterFeatureCallback()`. These pointers we then obtain as input parameters

in the callback function and thus can identify to which `CCamera` and `CFeature` the callback belongs.

At the end of the work with the camera we should unregister the callbacks and delete the helper `CFeature` class instances:

```
void CCamera::UnregisterFeatureCallbacks()
{
    for (size_t i=0; i<m_FeatureVector.size(); i++)
    {
        CFeature* pFeature = m_FeatureVector.at(i);
        m_pDevice->RegisterFeatureCallback(pFeature->m_FeatureId, NULL);
        delete pFeature;
    }
    m_FeatureVector.clear();
}
```

As you can see, the unregistration is done by specifying the `NULL` as the callback function for the feature.

The code above is enough for the first case of changes mentioned — when the change is a *result of change of other feature*. This is handled directly by the feature dependency mechanism and the callback is called as a direct result of the change (that means from the same thread, which made the change). However, our 2 features do not fall to this category, their value is changed independently on Simplon actions (the second mentioned category) and Simplon does not have any indication that the value was changed. For this reason, we must introduce an additional mechanism, called *polling*.

3.7.1. Polling non-cached features

As is explained in the Feature properties chapter, the features which change their values independently on Simplon are marked as *non-cached* features. This means the value must be always read from the hardware, it cannot be cached to speed up the reading. Simplon does not get any notification about the change of such feature, so there is no other way than read the value in a loop with a certain period; this is called *polling*. For example the `DeviceTemperature` is enough to read every 10 seconds, because it is changing only slowly, while the `LvDeviceUpTime` changes literally every microsecond, so there could be need to read such value with a high frequency. However, a periodic reading might cause unexpected load, for example on a GigE camera every read of the feature requires sending and receiving a packet, also a hardware action might be needed on the camera itself, so reading a feature with a high frequency may have significant side effects.

Non-cached features usually provide a recommended polling time period for the feature read. In Simplon it can be read by the following code:

```
LvFeature Feature = LvDevice_DeviceTemperature;

int32_t Cached;
pDevice->GetInfo(Feature, LvFtrInfo_IsCached, &Cached);

int32_t PollingTime;
pDevice->GetInfo(Feature, LvFtrInfo_PollingTime, &PollingTime);
```

Note that some features can return the `PollingTime = 0`; this means the feature is changing so rapidly that it does not make sense to suggest any polling time. Other features may return `−1`, this means the polling time is not defined.

The polling itself can be done directly by your application. But namely in case the application works with multiple features, it might be useful to utilize the feature callbacks, as already explained. To assure the feature callbacks are called for non cached features, it is just enough if the application extends the code explained in the previous chapter by adding a call of the `LvModule::Poll()` function in a loop or starting the polling thread using the `LvModule::StartPollingThread()`.

In the first case for example a system timer can be used, so that the polling happens in the main thread:

```
CMainWindow::OnTimer()
{
    pDevice->Poll();
}
```

A question is in what time period the `LvModule::Poll()` function should be called.

The answer is not trivial; theoretically it should be a minimum of recommended polling times of all features involved, but as some features may return PollingTime = 0, your application should anyway somehow determine a reasonable polling interval. For example if the purpose of the polling is only to update values on screen for human reading, then a 300 ms interval might be enough.

Note that Simplon internally keeps track by each feature when was its last read and does not make a new callback until the polling time of the feature expires. So if for example the `DeviceTemperature` feature has polling 10 seconds (10000 ms), then even if the `LvModule::Poll()` function is called every 250 ms, the `DeviceTemperature` feature gets a callback only after the 10 second time elapses.

Note also that the `LvModule::Poll()` function is a method of LvModule, that means you need to call it for every module (LvSystem, LvInterface, LvDevice, ...) on which you need to get the feature callbacks.

An alternative way to implement polling is to start an additional thread, which calls the `LvModule::Poll()` function in a loop. Simplon offers a simple way how to do so, by calling the function `LvModule::StartPollingThread()`:

```
pDevice->StartPollingThread(300, false);
```

The first parameter is a polling interval in milliseconds. The second parameter enables to use single polling thread for multiple modules: If set to true, also the features in all children modules are polled. For example, if your application uses only one System module, then it is a parent of all other modules, so the polling on the System will be propagated to all modules from a single thread. If a module has started own polling thread, then it is excluded from the propagating.

It is important to know, that the feature callback is called in the thread, from which the `LvModule::Poll()` was called; in this case it means it is called from the polling thread, not from the application main thread. All the subsequent actions thus must be *thread safe*.

3.7.2. Feature device event

The third way how a feature can be changed from the remote device is that the remote device sends a special type of **event** to Simplon, called *feature device event*. Simplon parses this event into values of particular features and for each modified feature calls the feature callback. After your application returns from the callback, Simplon can process the next event and place new data to the features.

To enable Simplon to process the events, a special type of `LvEvent` module must be created: `LvEventType_FeatureDevEvent` and a thread on it started:

```
LvEvent* m_pFeatureDevEvent;
//...
pDevice->OpenEvent(LvEventType_FeatureDevEvent,
                   m_pFeatureDevEvent);
m_pFeatureDevEvent->StartThread();
```

Starting a thread assures the parsing of event data and calling the feature callbacks on these features. In this case it is important to really read the feature value in the callback, because after return from the callback function the features can be already filled with new values from the next event.

3.7.2.1. Capturing logs from PicSight GigE Smart

A typical example of the *feature device event* mechanism are the log messages from PicSight GigE Smart camera. In this case the `EventLvSmartAppLogMessage` feature is acting as a cache buffer between the remote device and your application. The log produced on the device is a stream of lines, which is cut to parts that fit to a network packet. Each received packet is then converted to an event on the receiver side. So the `EventLvSmartAppLogMessage` feature does not receive one log line, but rather a part of the continuous text, which can contain multiple log lines (lines separated by the LF character), but it can contain also only a part of one long log line. The application should pick up the contents of this feature and *append* it to an internal text buffer. This buffer then contains the complete lines of the log, which can be displayed.

There is also another important thing to keep on mind: Unfortunately the GenICam does not give a possibility to distinguish in the feature callback *from where* it was issued. Thus can happen, that due to the changes in other features, the `EventLvSmartAppLogMessage` feature can get a feature callback, which does not come from the *feature device event*. In such case reading the feature value and appending it to the text buffer would cause that the contents would be appended twice to the text buffer. A solution in this case is to check the `EventLvSmartAppLogTimestamp` value — this value is also delivered with each new *feature device event* and must be different for each event. So the application can remember its last value and if it is still the same, ignore the feature callback.

3.7.3. Important notes for feature callbacks

From the previous text you can see that the origin of the feature callback is not obvious; it can be a result of other feature change as well as it can be an event from another thread. It is important to keep the following rules:

- The feature callback should never set any other feature. Doing so can lead to recursions, which would be probably hard to diagnose and could cause unexpected behavior or even a program crash. Imagine a situation when a *Feature2* is dependent on *Feature1*. When the *Feature1* is changed, it causes a feature callback on *Feature2*, so if you change the value of the *Feature1* from this callback, you get a recursion. This is a simple example, in reality the recursion may occur through a more complex chain of feature dependencies.

- The feature callback should do its job fast; if there is something complex to do, it should pass the task to another thread and return. This requirement comes namely from the fact, that due to complex feature dependencies one feature change can cause feature callbacks on a lot of features.

- The feature callback must be *thread safe* and you should consider a fact, that it can be called from other than the application main thread. Typical problems may arise in GUIs, for example when you want to update a label on screen with a new feature value, many GUI environments do not permit to do so from other than the main thread and breaking this rule usually leads to strange program behavior and crashes.

For example in Simplon Explorer, a feature callback is registered to *all* features. If the callback is called, Explorer only sets a *Modified* flag in the class instance representing the feature. The main thread then uses a timer, which every 200 ms iterates through all feature class instances and updates on screen those features, which have the *Modified* flag set. An exception to this are the `EventLvSmartAppLogMessage` and `EventLvSmartSysLogMessage` features — by these features the value must be read inside the feature callback (see the explanation in previous chapter). The value is read and placed to a string queue. From this queue (protected by a critical section) the strings are picked up in the main thread (again on a timer), appended to the log buffer and displayed.

3.7.4. Feature callbacks in Simplon .Net Library

Instead of the callback function, an **event handler** is used in the Simplon .Net Class Library. Let's illustrate in on an example in C#. In this sample handler we expect to get events based on changes in 2 features: `EventLvTriggerDropped` and `EventLvSmartAppLogMessage`. In the first case we simply increment a counter of dropped triggers, in second case we concatenate the received string with log to the existing log contents. We check the timestamp to avoid duplicated strings in case the event does not have an origin upon arrival of a new log data. In the `LvFeatureChangedArgs` the event handler receives 2 values, which you specify when you register the feature callback. We use simply the feature enum value as a pUserParam, so that we can use a `switch` statement and parse the event according to which feature it belongs.

```
void FeatureChangedHandler(System.Object sender, LvFeatureChangedArgs e)
{
    try
    {
        switch ((int) e.pUserParam)
        {
            case (int) LvDeviceFtr.EventLvTriggerDropped:
            {
                m_iDroppedTriggerCount++;
                break;
            }
            case (int) LvDeviceFtr.EventLvSmartAppLogMessage:
            {
                String Val_EventLvSmartAppLogMessage = "";
                Int64 Val_EventLvSmartAppLogTimestamp = 0;
                m_pDevice.GetString(LvDeviceFtr.EventLvSmartAppLogMessage,
                                    ref Val_EventLvSmartAppLogMessage);
                m_pDevice.GetInt(LvDeviceFtr.EventLvSmartAppLogTimestamp,
                                 ref Val_EventLvSmartAppLogTimestamp);
                if (m_LastLogTimestamp != Val_EventLvSmartAppLogTimestamp)
                {
                    m_LastLogTimestamp = Val_EventLvSmartAppLogTimestamp;
                    m_sSmartAppLog += Val_EventLvSmartAppLogMessage;
                }
                break;
            }
        }
    }
    catch (LvException)
    {
        // do not use GUI (display message) here,
        // it runs in different thread
    }
}
```

When we have a handler prepared, we can add this handler to `LvDevice::OnFeatureChanged` event:

```
m_pDevice.OnFeatureChanged += new LvFeatureChangedHandler(FeatureChangedHandler);
```

And we must register the features, which we want to monitor (note the usage of the feature enum value as the pUserParam):

```
m_pDevice.RegisterFeatureCallback(LvDeviceFtr.EventLvTriggerDropped,
     true, (IntPtr)LvDeviceFtr.EventLvTriggerDropped, (IntPtr)null);
m_pDevice.RegisterFeatureCallback(LvDeviceFtr.EventLvSmartAppLogMessage,
    true, (IntPtr)LvDeviceFtr.EventLvSmartAppLogMessage, (IntPtr)null);
```

This would already be enough for catching changes in the features based on changes in other features, on which are these features dependent. But the `EventLvTriggerDropped` and `EventLvSmartAppLogMessage` are features, the values of which are changed by the `feature device events`, so we must create the `LvEvent` module representing the feature device events and start a thread on it (already described in one of the previous chapters):

```
private LvEvent m_pFeatureEvent = NULL;
//...
m_pDevice.OpenEvent(LvEventType.FeatureDevEvent,
                    ref m_pFeatureEvent);
m_pFeatureEvent.StartThread();
```

And the mechanism of sending the feature device events must usually be setup also on the device:

```
m_pDevice.SetEnum(LvDeviceFtr.EventSelector,
                  (UInt32)LvEventSelector.LvTriggerDropped);
m_pDevice.SetEnum(LvDeviceFtr.EventNotification,
                  (UInt32)LvEventNotification.On);
m_pDevice.SetEnum(LvDeviceFtr.EventSelector,
                  (UInt32)LvEventSelector.LvSmartAppLog);
m_pDevice.SetEnum(LvDeviceFtr.EventNotification,
                  (UInt32)LvEventNotification.On);
```

Note that the feature callback can be removed by using `false` as the second parameter:

```
m_pDevice.RegisterFeatureCallback(LvDeviceFtr.EventLvTriggerDropped,
     false, (IntPtr)null, (IntPtr)null);
m_pDevice.RegisterFeatureCallback(LvDeviceFtr.EventLvSmartAppLogMessage,
     false, (IntPtr)null, (IntPtr)null);
```

3.8. Using the lv.simplon.ini library

The **lv.simplon.ini** library is a helper library for operating system independent reading and writing INI files. Its usage is quite simple and the description is available in the *Simplon reference guide*.

The following code snippet shows how to read values of various types:

```
LvHIniFile hIni = LvIniOpen();
LvIniLoad(hIni, "C:\\Data\\MyConfig.ini");
int iBuffers = LvIniGetInteger(hIni, "Settings",
                               "NumberOfBuffers", iBuffers);
bool bShowHidden = LvIniGetBool (hIni, "Settings",
                                 "ShowHiddenFeatures", 0)!=0;
char szCaption[256];
LvIniGetString (hIni, "Settings", "Caption", "",
                szCaption, sizeof(szCaption));
LvIniClose(hIni);
```

The following code snippet writes the configuration to a file:

```
LvHIniFile hIni = LvIniOpen();
LvIniLoad(hIni, "C:\\Data\\MyConfig.ini");
LvIniSetInteger(hIni, "Settings", "NumberOfBuffers", iNumberOfBuffers);
LvIniSetBool(hIni, "Settings", "ShowHiddenFeatures", (int)bShowHidden);
LvIniSetString (hIni, "Settings", "Caption", szCaption);
LvIniSave(hIni, "C:\\Data\\MyConfig.ini");
LvIniClose(hIni);
```

Note that in order to preserve the other contents of the INI file it must be first *loaded*.

4. Simplon Features Reference

The GenTL features and Device Remote features are described in detail in the PicSight GigE Manual and CheckSight Manual The Simplon library adds own features to the System, Interface, Device and Stream modules. The Renderer modules is solely implemented in the Simplon library (not provided by the GenTL producer) and so all the Rendeder features are coming from the Simplon library.

4.1. System

Display Name `LvSystemDisplayName` (String)

Returns a user readable name of the system.

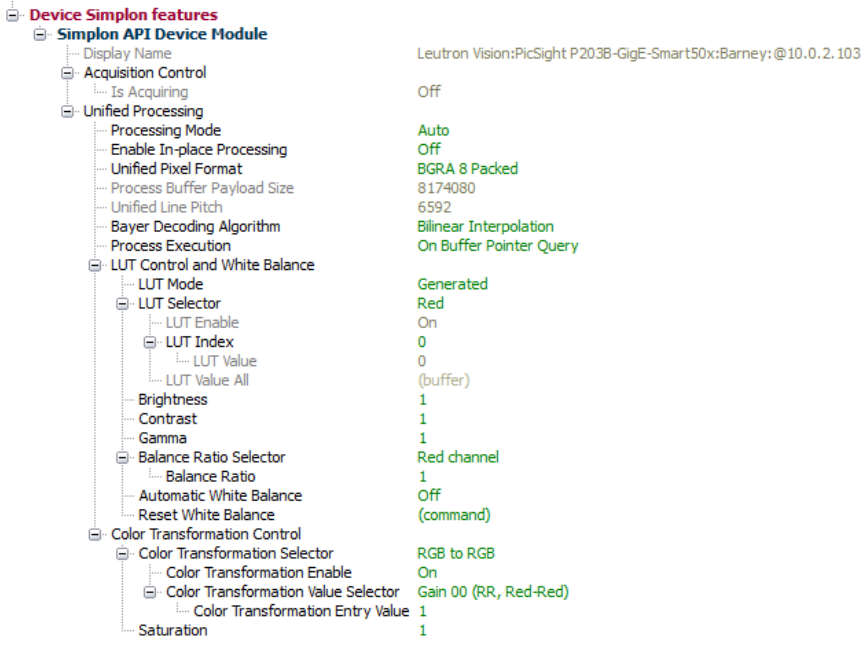
4.2. Interface

Display Name `LvInterfaceDisplayName` (String)

Returns a user readable name of the interface.

4.3. Device

The features added to the Device module are related mostly to the unified image preprocessing, described in the chapter Section 3.3, “Unified image preprocessing” [p. 62].



Display Name `LvDeviceDisplayName` (String)

Returns a user readable name of the device.

Is Acquiring `LvDeviceIsAcquiring` (Boolean)

Returns true if the acquisition was started. Note that this feature does not tell whether the images are actually delivered to the output buffer queue; it simply informs that your application is between the AcquisitionStart and AcquisitionStop actions.

Processing mode

`LvUniProcessMode` (Enumeration)

The UniProcessing provides unified API for image preprocessing, which is done either on the device itself, if it is supported by the hardware, or by software, if not. The preprocessing includes Bayer decoding or pixel format conversion, application of LUT and Color Correction.

Values:
- `HwOnly` - The processing is done only in case it is available directly on the hardware (device). The images will be delivered to the output buffer queue already processed.
- `SwOnly` - The processing will be done by software even if the hardware could support the operation. The software processing is done when the buffer is passed to the output buffer queue (or later - see LvUniProcessExecution).
- `Auto` - The processing will be done by hardware and by software will be processed only the part, which is not possible to do on hardware. Note that if the Bayer decoding is done by software (this happens when you select an undecoded Bayer pixel format as the device PixelFormat), the LUT must be then also done by software, even if it is available in hardware; that's because it must be applied after the Bayer decoding.

Enable In-place Processing

`LvUniProcessEnableInPlace` (Boolean)

If possible, the software image preprocessing will be preferably done in the same image (not to another buffer). This is possible only in case the preprocessing does not change the pixel format, that means the LvUniPixelFormat must be equal to PixelFormat (for example the Bayer decoding is not done by software).

Unified Pixel Format

`LvUniPixelFormat` (Enumeration)

If the image preprocessing is enabled, this is the desired pixel format, to which the image is to be converted. Only monochrome and RGB/BGR color pixel formats are supported. The processing chain is set so that:
- if the `PixelFormat` is undecoded Bayer, the Bayer decoding to desired `LvUniPixelFormat` is included
- otherwise if the `PixelFormat` is not equal to `LvUniPixelFormat`, a pixel format conversion is included.

Process Buffer Payload Size

`LvUniProcessPayloadSize` (Integer)

Returns the size needed for the processing output buffer, which is to be used when the in-place processing is not enabled or not possible. Normally is this buffer allocated automatically for each acquisition buffer, but your application can also provide your own buffers and this feature gives the size of the buffers needed.

Unified Line Pitch

`LvUniLinePitch` (Integer)

The line increment of the process buffer, if the processing is active, or of the source buffer, if processing is not active. To

access the image regardless whether it was processed to the process buffer or not, you need 5 independent values:

- Pointer to the data - use LvUniBase feature of the Buffer, which points either to the acquired image (if no processing was done), or to the processed image (if it was processed).
- Width and height - these are the same for the acquired and processed image, so use the Width and Height from the remote device, or ChunkWidth and ChunkHeight if these can change during the acquisition.
- Pixel format - use LvUniPixelFormat - if this is different from the PixelFormat then processing is done, so LvUniPixelFormat is always correct.
- Line pitch - use LvUniLinePitch, which returns proper line pitch of the buffer, to which the LvUniBase pointer points.

Note that the above is valid only in case the processing can be successfully done (for example the source image is not in unsupported `PixelFormat`) and is not disabled (for example by `LvUniProcessExecution=OnExplicitRequest`).

Bayer Decoding Algorithm

`LvUniBayerDecoderAlgorithm` (Enumeration)

Selects the Bayer array decoding method for the software processing. This does not apply to the hardware Bayer decoding on the device, which is usually fixed to one method.

Values:

- `NearestNeighbour` - Fastest decoding, giving the worst results, enables also decoding to a monochrome pixel format.
- `BilinearInterpolation` - Fast common decoding, enables also decoding to a monochrome pixel format.
- `BilinearColorCorrection` - Decoding with quick enhacements on edges.
- `PixelGrouping` - Slower decoding, giving very good results.
- `VariableGradient` - Slowest decoding, giving the best results.

Process Execution

`LvUniProcessExecution` (Enumeration)

Defines the point, when the software image processing of the buffer is done. You may need to define this point in case you do not need all the images to be processed. Note that this applies only to the software processing; the hardware processing is done on the remote device always.

Values:

- `OnBufferPtrQuery` - The SW image processing is delayed to the time the application asks for the `LvBuffer_UniBase` or `LvBuffer_ProcessBase` pointer or for the `LvipImgInfo` data. This enables to the application to skip the processing in case it is not needed. If this is queried several times for the same image, the processing is done only once.
- `OnPopFromQueue` - The SW image processing is done always - at the moment the buffer is popped from the output buffer queue, before delivering it to the application.

- `OnExplicitRequest` - The SW processing is not done automatically, but must be explicitly done by the `ExecProcess` command feature of the Buffer.

LUT Mode

`LvUniLUTMode` (Enumeration)

Selects the LUT control mode. The mode determines, if the LUT can be directly modified by the application, or if the LUT is to be reserved for implementation of white balance, gamma, brightnesss and contrast - in such case the LUT is filled with precalculated values by Simplon library and cannot be directly modified.

Values:
- `Direct` - In this mode the LUT is controlled directly.
- `Generated` - In this mode the LUT is controlled through the higher level features, such as brightness, contrast, gamma or white balance.

LUT Selector

`LvUniLUTSelector` (Enumeration)

This selector selects for which LUT is applied `LvUniLUTIndex` / `LvUniLUTValue`. In case of monochrome image the LUT has only one array = `Luminance`. In case of color images, the LUT consists of 3 arrays, for Red, Green and Blue.

Values: `Luminance`, `Red`, `Green`, `Blue`

LUT Enable

`LvUniLUTEnable` (Boolean)

Enables the LUT in the processing. When disabled, the LUT does not lose its values, the disabled LUT is substitued by a linear LUT, and when enabling the LUT, the original values are retained.

LUT Index

`LvUniLUTIndex` (Integer)

Index of the element to be accessed in the selected LUT via the `LvUniLUTValue` feature. Note that accessing the whole LUT by this approach can be very time consuming, namely on GigE cameras. If possible, it is better to use the `LvUniLUTValueAll` or Simplon dedicated LUT functions.

LUT Value

`LvUniLUTValue` (Integer)

Value of the element for the current `LvUniLUTIndex` in the selected LUT. Note that accessing the whole LUT by this approach can be very time consuming, namely on GigE cameras. If possible, it is better to use the `LvUniLUTValueAll` or Simplon dedicated LUT functions.

LUT Value All

`LvUniLUTValueAll` (Buffer)

This feature enables to get/set the entire content of the selected LUT in one block access. Beware that the LUT buffer structure is vendor and model dependent, so take care if your application is exptected to work with various types od devices or devices from various vendors.

Brightness

`LvUniBrightness` (Float)

Brightness of the image. It is realized by the LUT. Values under 1.0 means darker than original, above 1.0 lighter than the original. The `LvUniLUTMode` must be `Generated`, in order to enable this feature.

Contrast `LvUniContrast` (Float)

Contrast of the image. It is realized by the LUT. Values under 1.0 means lower contrast than original, above 1.0 higher contrast than the original. The `LvUniLUTMode` must be `Generated`, in order to enable this feature.

Gamma `LvUniGamma` (Float)

Gamma correction of the image. It is realized by the LUT. Values under 1.0 make the middle tones darker, above 1.0 lighter. The `LvUniLUTMode` must be `Generated`, in order to enable this feature.

Balance Ratio Selector `LvUniBalanceRatioSelector` (Enumeration)

Selects which color channel will be accessed by the `LvUniBalanceRatio` feature. The `LvUniLUTMode` must be `Generated`, in order to enable this feature. Values: `Red`, `Green`, `Blue`

Balance Ratio `LvUniBalanceRatio` (Float)

The white balance factor to be applied on the selected color channel. The selected color channel of all pixels will be multiplied by this value (not directly, but through the precalucated LUT). If the value is < 1.0, the saturated pixels will become gray (white is no more white). Thus it is better if all 3 factors are greater than or equal to 1.0.

Automatic White Balance `LvUniBalanceWhiteAuto` (Self-clearing Enumeration)

Selects the action for automatic white balance calculation. Currently only the option Once is available. Setting this option causes the following:

- If there is already acquired image available, the white balance factors are calculated from this image and LUT is updated to reflect the changes.
- If there is no image acquired yet, an internal flag is set and the calculation is done when the image is acquired.

Note that the enumeration is self-clearing, that means its value is automatically changed to Off, when the white balance calculation is finished. The newly calculated white balance is applied to to newly acquired images, not to the existing ones, unless you explicitly call the ExecProcess command for the already acquired buffers. At the time of calculation the camera should look at a neutral grey (not white) object, which should fill the whole image area. Making white balance from normal image can bring less satisfactory results.

Values:

- `Off` - Automatic white balance mode off - the automatic white balance is not applied.

- `Once` - Automatic white balance mode once - the white balance factors are once adjusted, then switches the enumeration back to the Off value.

Reset White Balance `LvUniBalanceWhiteReset` (Command)

Sets all the white balance factors (`LvUniBalanceRatio`) to 1. The advantage of this feature in comparison with setting the 3 factors to 1 is that the LUT is updated only once, so it is faster.

Color Transformation Mode `LvUniColorTransformationMode` (Enumeration)

Selects the Color Transformation matrix control mode. The mode determines, if the matrix can be directly modified by the application, or if the matrix is to be reserved for implementation of the Saturation or other higher level features - in such case the matrix is filled with precalculated values by Simplon library and cannot be directly modified.

Values:
- `Direct` - In this mode the Color Transformation matrix can be controlled directly.
- `Generated` - In this mode the Color Transformation matrix is set through the higher level features, such as the Saturation.

Color Transformation Selector `LvUniColorTransformationSelector` (Enumeration)

Selects which color transformation module is controlled by the color transformation features. It also gives particular meaning to individual color transformation gains. Values: `RGBtoRGB` - currently the only Color Transformation matrix type.

Color Transformation Enable `LvUniColorTransformationEnable` (Boolean)

Enables the Color Transformation in the processing. When disabled, the Color Transformation matrix does not lose its values; when enabling it, the original values are retained.

Color Transformation Value Selector `LvUniColorTransformationValueSelector` (Enumeration)

Selects the cell of the Color Transformation matrix to be accessed by LvUniColorTransformationValue.

Values: `Gain00`, `Gain01`, `Gain02`, `Gain10`, `Gain11`, `Gain12`, `Gain20`, `Gain21`, `Gain22`

Color Transformation Entry Value `LvUniColorTransformationValue` (Float)

The value of the selected cell of the Color Transformation matrix.

Saturation `LvUniSaturation` (Float)

Sets the Color Correction matrix according to specified saturation. The saturation set to 0 causes a conversion to greyscale, 1.0 leaves the image identical, 2.0 emphasizes the colors.

4.4. Stream

Display Name `LvStreamDisplayName` (string)

Returns the display name of the stream.

Calculate Payload Size `LvCalcPayloadSize` (integer)

Returns the payload size (size of buffer to hold the image data). If the payload size is not provided by the stream or device, it is calculated, so this feature returns always a valid value.

Postpone Queue Buffers `LvPostponeQueueBuffers` (integer)

Number of buffers to be kept postponed before returning to the input buffer pool. This is useful when you need to keep last `N` acquired images, for example in order to be able to repaint `N` tiles of last acquired images on the screen. If `LvPostponeQueueBuffers` is > 0, then an additional queue on `N`-size is inserted between the `LvBufferQueue()` function and actual placement of the buffer to the input buffer pool.

Await Delivery Limit `LvAwaitDeliveryLimit` (integer)

Limit for images in the output buffer. Applicable only if the event thread is running - then if there is more than this number of buffers in the output queue, the oldest buffers are discarded and returned to input buffer pool. This is useful in case the application is not able to process all the images in time.

Auto Allocate Process Buffers `LvAutoAllocateProcessBuffers` (boolean)

Enable the auto allocation of process buffers. The process buffers are allocated only if they are needed for the image processing or conversion. You can disable the automatic buffer alocation and provide own buffers, using the `LvBufferAttachProcessBuffer()` function.

Preallocate Process Buffers `LvPreallocateProcessBuffers` (command)

Preallocates all the process buffers, even if it is not yet sure if they will be needed. With this command you can avoid time delays when allocating the buffers during the acquisition.

Delivered Frames `LvNumDelivered` (integer)

Number of acquired frames since last acquisition start. It is equivalent to the GenTL `STREAM_INFO_NUM_DELIVERED` info.

Underrun Frames `LvNumUnderrun` (integer)

Number of lost frames due to input buffer pool underrun. It is equivalent to the GenTL `STREAM_INFO_NUM_UNDERRUN` info.

Announced Buffers `LvNumAnnounced` (integer)

Number of announced buffers. It is equivalent to the GenTL `STREAM_INFO_NUM_ANNOUNCED` info.

Queued Buffers

`LvNumQueued` (integer)

Number of buffers currently in the input pool. It is equivalent to the GenTL `STREAM_INFO_NUM_QUEUED` info.

Buffers Awaiting Delivery

`LvNumAwaitDelivery` (integer)

Number of buffers currently in the output queue. It is equivalent to the GenTL `STREAM_INFO_NUM_AWAIT_DELIVERY` info.

Is Grabbing

`LvIsGrabbing` (boolean)

Flag indicating whether the acquisition engine is started or not. This is independent from the acquisition status of the remote device. It is equivalent to the GenTL `STREAM_INFO_IS_GRABBING` info.

4.5. Renderer

Automatic Display

`LvAutoDisplay` (boolean)

If set, the image is automatically displayed before it is passed to the supplied callback. This is functional only in case the Event thread is started.

Render Type

`LvRenderType` (enumeration)

Controls way how the acquired images are rendered on the screen.

Values:
- `FullSize` - Renders the acquired image in full size.
- `ScaleToFit` - Renders the acquired image to fit into the window.
- `ScaleToSize` - Renders the acquired image scaled to required size.
- `ScaleToTiles` - Renders the acquired images in tiles.

Note that all the Scale- options require scaling capability of the display and might not be supported in all operating systems.

Offset X

`LvOffsetX` (integer)

Sets the horizontal offset of the image to be rendered, i.e. the distance from the left edge of the display window.

Offset Y

`LvOffsetY` (integer)

Sets the vertical offset of the image to be rendered, i.e. the distance from the top edge of the display window.

Width

`LvWidth` (integer)

Sets the width of the rectangle to which the image is to be rendered. Note that if the `LvIgnoreAspectRatio` feature is `False`, the real image width can be smaller, in order to keep the aspect ratio.

Height

`LvHeight` (integer)

Sets the height of the rectangle to which the image is to be rendered. Note that if the `LvIgnoreAspectRatio` feature is `False`, the real image height can be smaller, in order to keep the aspect ratio.

Automatic Tile Calculation `LvAutoTileCalculation` (boolean)

When set to True, the tile sizes and positions are calculated automatically. When the `LvColumns` and/or `LvRows` are 0, also the number of columns and/or rows is calculated automatically.

Number of Tiles `LvNumberOfTiles` (integer)

Sets the number of tiles used for image rendering. Note that for the tile repaint is needed that the corresponding buffers are still in the application ownership; once the buffer is placed to the input buffer pool, it should not be accessed for paint anymore (see also `LvPostponeQueueBuffers`).

Display Columns `LvColumns` (integer)

Sets the number of columns used for image rendering. When the value is 0, the number of columns is calculated automatically.

Display Rows `LvRows` (integer)

Sets the number of rows used for image rendering. When the value is 0, the number of rows is calculated automatically.

Gap Between Tiles `LvTileGap` (integer)

Gap between the tiles in pixels.

Ignore Aspect Ratio `LvIgnoreAspectRatio` (boolean)

Allows to ignore the original aspect ratio while rendering the image, so the image can be scaled up/down in one dimension with different factor than in the other dimension.

Disable Scale Up `LvDisableScaleUp` (boolean)

Disables scaling the image up.

Disable Scale Down `LvDisableScaleDown` (boolean)

Disables scaling the image down.

Center Image `LvCenterImage` (boolean)

Centers the rendered image in the window.

5. The image preprocessing library

The image preprocessing library serves for fast image preprocessing done in the host memory. This preprocessing includes:

- **Bayer array decoding**.
- Applying a **Lookup Table** (LUT). The LUT can be applied either in a separate step, or faster as a part of other processing, so many functions take a handle to the LUT as an optional parameter.
- **Pixel format conversion**.
- **RGB color correction** by applying 3x3 matrix to the R,G,B vector of the pixel. By the RGB color correction for example the saturation as well as the white balance of the color image can be adjusted.
- Applying a **3x3 convolution matrix**, by which a sharpening or edge detection can be implemented.
- **Shading correction**, where a black and white reference images form an offset and gain individually for each pixel in the image.
- **Extracting a rectangle** (region of interest) from an image.
- **Rotation**, **mirroring**, deinterlacing, reversing lines.
- **Saving and loading** bitmaps to BMP, TIFF and JPEG files.

The library is optimized for speed and a stress is put on the ease of use and its robustness. It exists in 2 versions: as the **DLL version** with a plain C API, and as the **Class Library for the Microsoft .Net Framework** 2.0 and higher.

5.1. Principles of Usage

5.1.1. Image Descriptor

The main object needed for the image manipulation is a *descriptor* of the image parameters (like width, height, pixel format, line increment etc.), which also contains a pointer to the image data buffer. In the DLL version this descriptor is a structure `LvipImgInfo`, while in the .Net version it is represented by the `LvipImage` class. The descriptor contains everything needed to work with the image and thus fully represents the image.

The most common operation is to take image pixels from the *source* image and place the result of the processing to the *destination* image; thus most processing functions take the descriptor of the source and destination images as the parameters (in case of .Net version the processing functions are methods of the `LvipImage` class, which also represents the source image for the operation, so only the destination image class is passed as a parameter). In some cases it is possible to place the result of the processing to the same image (so called **in-place processing**) — this is generally possible when

- the source and destination image sizes are equal (not true for example for rotation)
- the pixel format after the operation is the same (not true for example for Bayer decoding)
- the same source pixel is not used multiple times in the destination pixels calculation (not true for example for applying a convolution matrix)

The in-place processing is used, when the destination image is not specified, that means instead of the destination image the `NULL` is used in the DLL version, or `nullptr` in the C++ .Net, or `null` in the C# .Net or `Nothing` in the Visual Basic .Net.

5.1.2. Image Buffer Allocation

The image descriptor contains a pointer to the image data, i.e. a buffer with pixel values. The pointer can be used in 3 ways, which *should not be mixed*:

- Let the ImgProcLib allocate and deallocate the buffer(s) for the image, using the `LvipAllocateImageData()` and `LvipDeallocateImageData()` functions in the DLL version, or `LvipImage::AllocateImageData()` and `LvipImage::DeallocateImageData()` methods in the .Net class library. This is the recommended way; when the image parameters (width, height, pixel format and flags) are set, the allocation is simple function or method call. Note that in the DLL version you are responsible for calling the deallocation function whenever you want to change the pointer value (typically before allocating a new buffer when the image parameters change). In the .Net Class Library the deallocation is called automatically at the class destructor and also whenever the image parameters change (if the change has an influence to the needed buffer size). However memory allocation must still be coded explicitly even in the .Net version.

- Set the image pointer to point to a buffer which is managed by some other owner. Typically, when the image is acquired to a DMA buffer, it would be wasting time to copy it to another buffer. In this case the `LvipImgAttr_NotDataOwner` flag in the DLL version or `lvipImgFlags::AttrNotDataOwner` in the .Net version must be set, so that the pointer is excluded from attempts to deallocate it.

- Allocate and deallocate the buffers in your application and assign the pointers to the image descriptor. In such case the buffer management is solely on your code and you should take care about allocating an appropriate size. After you call the `LvipInitImgInfo()`, the `LvipGetImageDataSize()` can be used for obtaining the size of the buffer (*LvipImage::ImageDataSize* property in the .Net version). Also in this case the `LvipImgAttr_NotDataOwner` flag in the DLL version or `lvipImgFlags::AttrNotDataOwner` in the .Net version must be set, so that the pointer is excluded from attempts to deallocate it by ImgProcLib.

5.1.2.1. Automatic Buffer Reallocation

To make the programming easier, most of the functions enable to allocate or reallocate automatically the image buffer(s) in case it is needed. For this feature the `LvipOption_ReallocateDst` flag in the DLL version or the `lvipImgFlags::ReallocateDstImg` flag in the .Net version must be used. In such case it is enough to supply an empty image descriptor for the destination image and the function will fill it with proper parameters and will allocate a buffer when it is called for the first time. When the function is called the next time, it only verifies, if the image descriptor is compatible with the destination image, and if so, no new allocation is done; this assures that the time is not wasted by repeated reallocation. In the DLL version be sure to deallocate the image data buffer (by the `LvipDeallocateImageData()` function) when no more needed; in the .Net version you can rely on the `LvipImage` class destructor, which deallocates the buffer automatically.

Note that the `LvipOption_ReallocateDst` flag can be combined with `LvipImgAttr_DWordAligned` or `LvipImgAttr_QWordAligned` flags, so that the line increment has the desired alignment (these flags are passed to the `LvipInitImgInfo()` function). In the .Net Class Library, the flags are `lvipImgFlags::DWordAligned` and `lvipImgFlags::QWordAligned`

5.1.3. Lookup Table (LUT)

The Lookup Table is a conversion table, through which each pixel value is translated. With the LUT various actions can be implemented, like the gamma correction, brightness and contrast modification etc. When 3 LUTs are applied separately to R, G and B channels of a color image, a white balance can be adjusted.

The LUT size corresponds to the range of pixel values, that means:

- the LUT for 8–bit pixel format has the size of 256 bytes,
- the LUT for 10–bit pixel format has the size of 1024 words and
- the LUT for 12–bit pixels format has the size 4096 words.

There are no other LUT sizes supported in ImgProcLib. The 8-bit LUT can be used for 8-bit monochrome images, 24-bit and 32-bit RGB color images. It has 3 tables, for R, G and B channels of the pixel; when applied to a monochrome image, the G (green) table is used. The 10-bit LUT can be used for 10-bit monochrome images only and the 12-bit LUT can be used for 12-bit monochrome images only. Other pixel formats cannot use the LUT; it is recommended to either acquire the images in one of these supported formats, or convert the image to an appropriate pixel format before the image is processed.

In case you prefer comfort in programing, you can use the unified LUT (see the `LvipLutType_Uni` type), which internally keeps 3 LUTs — 8–bit, 10–bit and 12–bit. All LUT operations are applied to all 3 LUTs and when you pass such LUT as a parameter to a processing function, automatically the appropriate LUT is selected according to the pixel format of the processed image. The tax for this comfort is more CPU memory occupied.

Applying LUT is quite common task and it is often more effective to apply it during some other pixel operation, rather than loosing time by iterating the pixels again separately. For this reason the LUT can be passed as an optional parameter to many image processing functions.

For some Bayer decoding methods a LUT with additional precalculated tables is needed; this precalculation enables faster decoding than if the calculation would have to be done for each pixel. You can enable the creation of the additional tables by an option when the LUT is created. The LUT with precalculated values occupies more memory.

In the DLL version the LUT is represented by a handle of `LvipHLut` type; in the .Net version the LUT is represented by the `LvipLut` class.

The library always creates 3 **global LUTs** (8–bit, 10–bit and 12–bit). When you specify the `LVIP_LUT_GLOBAL` constant as a handle to the LUT, the function selects and uses one of these 3 LUTs. In the .Net version the global LUT is represented by the `LvipLut` class instance, created with the `lvipLutTypes::TypeGlobal` attribute in the constructor. The global LUT is easier to use; you need not to take care about the LUT creation in your code; however, as it is global, it is not suitable in case you work simultaneously with multiple images or in a multithreading environment.

In case you do not want to use a LUT in a function, pass `NULL` as the parameter for the LUT handle (the DLL version), or `nullptr` in the C++ .Net, or `null` in the C# .Net or `Nothing` in the Visual Basic .Net. Note that some of the Bayer decoding functions require always a LUT, so there you must pass a valid LUT handle or class.

5.2. Troubleshooting

When something does not work as expected, please first check in your code if you have proper error handling done in your application - see the *Error Handling* for details. If you ignore some error status, the behavior of your application may be other than expected. The library logs the errors in the Simplon log file, see Section 3.6, “Simplon log output” [p. 68].

5.3. The DLL version

The DLL version provides a plain C API.

The library is supplied with a C/C++ header file `lv.simplon.imgproc.h`. All published identifiers in the library are prefixed with `Lvip` (function names, structures) or `LVIP_` (defines).

5.3.1. Image Info Descriptor

Each image handled by the library must be described by the `LvipImgInfo` structure. This structure is defined as follows:

```
typedef struct
{
  uint32_t InfoSize;
  uint32_t Width;
  uint32_t Height;
  uint32_t PixelFormat;
  uint32_t Flags;
  uint32_t BytesPerPixel;
  uint32_t LinePitch;
  uint8_t* pData;
  uint8_t* pDataR; // red
  uint8_t* pDataG; // green
  uint8_t* pDataB; // blue
} LvipImgInfo;
```

- `InfoSize` - should be set to the `sizeof(LvipImgInfo)`. This member may be used in the future versions for the compatibility check.
- `Width`, `Height` - width and height in pixels.
- `PixelFormat` - pixel format - must be set to one of the values defined in the `LvPixelFormat` enum.
- `Flags` - flags indicating other features and options. Can be combined by the binary `or` operator.
  - `LvipImgAttr_NotDataOwner` - image data are not owned, belong to another object, so do not deallocate them when the image info is cleared.
  - `LvipImgAttr_BottomUp` - orientation - first line in the image buffer is the bottom line. By default the images are top-down.
- `BytesPerPixel` - bytes per pixel
- `LinePitch` - bytes per line (also sometimes called *line increment*)
- `pData` - pointer to image data (in case color planes are not used)
- `pDataR` - pointer to the red color plane image data (in case color planes are used)
- `pDataG` - pointer to the green color plane image data (in case color planes are used)
- `pDataB` - pointer to the blue color plane image data (in case color planes are used)

Note that the color planes are not supported by the current version of the library.

In the DLL version the `LvipImageInfo` structure is not protected so you can change any member of it. However, keep on mind that the parameters must correspond. For example when you change the pixel format, the line pitch may need a change as well. Thus it is highly recommended to prefer the `LvipInitImgInfo()` for setting the image parameters rather than setting the parameters directly.

The library provides functions for initializing the `LvipImgInfo` structure:

- `LvipInitImgInfo()` - initializes the structure from width, height, pixel format and flags. Note that this function sets the image data pointers to NULL, so if the buffers were already allocated, be sure to deallocate them before this function call.

The line pitch of the image depends on the following flags: `LvipImgAttr_DWordAligned` - line increment should be aligned to 4 bytes, `LvipImgAttr_QWordAligned` the line increment should be aligned to 8 bytes.

- `LvipBmpInfoToImgInfo()` - initializes the structure from Windows `BITMAPINFO` structure. Note that `LvipImgInfoToBmpInfo()` does the opposite - converts the Windows `BITMAPINFO` structure to `LvipImgInfo`.
- `LvipAllocateImageData()` - allocates appropriate memory buffer for the image
- `LvipDeallocateImageData()` - deallocates the allocated memory, if the image buffer is owned by this image info and sets the data pointer(s) to NULL.

5.3.2. Lookup Table

The LUT is not passed as a simple array, but rather as a handle of `LvipHLut` type to an internal structure, which is allocated by the `LvipAllocateLut()` function and freed by the `LvipFreeLut()` function. The reason for that is that 6 types of LUTs exist: 8-bit, 10-bit and 12-bit and the same triplet with extensions for Bayer decoding. The LUTs which are to be used with the Bayer decoding, must have special extensions - to create such a LUT, use the `LVIP_LUT_BAYER` flag in the `LvipAllocateLut()` function. The extensions include additional precalculated tables for faster 3x3 bilinear Bayer decoding.

The LUT can be filled by values by the `LvipResetLut()`, `LvipSet8BitLut()`, `LvipSet10BitLut()` and `LvipSet12BitLut()` functions, furthermore functions like `LvipAddWbToLut()` and `LvipAddGammaToLut()` can be used to modify the LUT.

5.3.3. Sample code

In the following sample we will demonstrate something similar to the automatic image preprocessing in Simplon: if the pixel format is Bayer array, we will decode the image to a BGR pixel format, then we will apply LUT calculated from white balance, gamma, brightness and contrast and finally we will apply a color correction. The processed image is save to TIFF. This illustrates a core usage of the image processing library.

First we need a helper function, determining if the pixel format is Bayer. It is quite simple:

```
bool IsBayer(uint32_t PixelFormat)
{
    switch(PixelFormat)
    {
        case LvPixelFormat_BayerGR8:
        case LvPixelFormat_BayerRG8:
        case LvPixelFormat_BayerGB8:
        case LvPixelFormat_BayerBG8:
        case LvPixelFormat_BayerGR10:
        case LvPixelFormat_BayerRG10:
        case LvPixelFormat_BayerGB10:
        case LvPixelFormat_BayerBG10:
        case LvPixelFormat_BayerGR12:
        case LvPixelFormat_BayerRG12:
        case LvPixelFormat_BayerGB12:
        case LvPixelFormat_BayerBG12:
            return true;
    }
    return false;
}
```

The following function takes as a parameter a pointer to `LvBuffer`; from `LvBuffer` we can obtain the `LvipImgInfo` descriptor.

```
void ProcessAndSaveImage(LvBuffer* pBuffer)
{
    LvipImgInfo SrcImgInfo;
    pBuffer->GetImgInfo(SrcImgInfo);
    uint32_t FactorRed, FactorGreen, FactorBlue;
    LvipCalcWbFactors(&SrcImgInfo,
                      &FactorRed, &FactorGreen, &FactorBlue, 0);

    LvipHLut hLut;
    hLut = LvipAllocateLut(LvipLutType_UniBayer);
    LvipResetLut(hLut);
    LvipAddGammaToLut(950, hLut);
    LvipAddBrightnessAndContrastToLut(950, 1000, hLut);

    LvipAddWbToLut(FactorRed, FactorGreen, FactorBlue, hLut);

    LvipImgInfo DstImgInfo;
    memset(&DstImgInfo, 0, sizeof(DstImgInfo));
    DstImgInfo.StructSize = sizeof(DstImgInfo);

    if (IsBayer(SrcImgInfo.PixelFormat))
        LvipBdBilinearInterpolation(&SrcImgInfo, &DstImgInfo,
                                    LvPixelFormat_BGR8Packed,
                                    LvipOption_ReallocateDst, hLut);
    else
        LvipApplyLut(&SrcImgInfo, &DstImgInfo, hLut,
                     LvipOption_ReallocateDst);

    int32_t ColorMatrix[9];
    LvipSetSaturationMatrix (120, ColorMatrix, 0);
    LvipApplyRgbColorCorrection (&DstImgInfo, NULL, ColorMatrix,
                                 0, NULL);

    LvipSaveToTiff("C:\\Data\\Test\\Image.tif", &DstImgInfo, 0);

    LvipFreeLut(hLut);
    LvipDeallocateImageData(&DstImgInfo);
}
```

The `LvBuffer::GetImgInfo()` function returns `LvipImgInfo` image descriptor of the acquired image, we put it to the `SrcImgInfo`. If the image is Bayer encoded or in RGB color format, we will need the factors for correction of the white balance. These factors can be often calculated automatically from the image, using the `LvipCalcWbFactors()` function.

For applying the white balance, gamma, brightness and contrast we will need a LUT, which is to be created by the `LvipAllocateLut()` function. Here we use the `LvipLutType_UniBayer` type, which is universal for 8–, 10– and 12–bit mono formats as well as for 24–bit and 32–bit BGR. It also keeps precalculated values for Bayer decoding. This universality is paid by a higher memory consumption. The `LvipResetLut()` function resets the LUT to a linear 1:1 status. The following functions `LvipAddGammaToLut()`, `LvipAddBrightnessAndContrastToLut()` and `LvipAddWbToLut()` *add* the desired parameters to the LUT. The resulting LUT is either applied during the Bayer decoding, or directly, if the decoding is not applicable.

For converted image we need a destination image descriptor and buffer. For it we create a `DstImgInfo` structure, wipe it with zeros and set its size. In the following functions we utilize the `LvipOption_ReallocateDst` option, which fills the structure with appropriate parameters (width, height etc.) and allocates the buffer for image data.

If the source image is Bayer array encoded, we utilize the `LvipBdBilinearInterpolation()` function to convert it to color image in 24–bit BGR format. The LUT is applied during the conversion. If the image has other format, the `LvipApplyLut()` function applies the LUT to it.

For the saturation we need to utilize the 3x3 color correction matrix. The `LvipSetSaturationMatrix()` function fills the matrix with values corresponding to selected saturation and the `LvipApplyRgbColorCorrection()` function applies the matrix to the

destination image. Notice the `NULL` as the second parameter of this function call — this means the operation is *in-place*, that means the result is placed to the same buffer.

The converted image is saved to a TIFF file, using the `LvipSaveToTiff()` function.

And finally, the LUT and destination buffer are deallocated.

The sample code is simplified to make it clear. In a normal application the allocation and deallocation would be probably done in other parts of the program and the LUT and buffer would be reused for multiple image processing. This sample also does not solve any possible error states, for example the RGB color correction cannot be applied to an image with a monochrome pixel format. Also all the parameters, like the *brightness*, *contrast* etc., as well as the *file name*, would not be directly hardcoded like we have it here.

5.4. The .Net Class Library Version

The ImgProcLib .Net Class Library is a wrapper around the DLL version of the library; this wrapper provides a set of Common Runtime Language managed classes, which can be easily used from the compilers in the .Net Framework 2.0 and higher, like C++, C# and Visual Basic. The library provides the same functionality as the DLL version, but the DLL defines, types and functions are converted to classes for easier usage:

- The `LVIP_PIXEL_FORMAT_xxx` constants are represented by the `lvipPixelFormat` enumeration class.
- The `LVIP_IMG_xxx` flags are represented by the `lvipImgFlags` enumeration class.
- The `LVIP_FUNCT_xxx` flags are represented by the `lvipFunctFlags` enumeration class.
- The `LVIP_LUT_xxx` flags are represented by the `lvipLutTypes` and `lvipLutSize` enumeration classes.
- The `LvipImgInfo` structure is represented by the `LvipImage` class. All the functions, which in the DLL version take a source image descriptor as a parameter, are the methods of the `LvipImage` class in the .Net version.
- The `LvipHLut` is represented by the `LvipLut` class. All the functions for manipulating with the LUT in the DLL version are the methods of `LvipLut` class in the .Net version. The **global LUT** is represented by an instance of `LvipLut`, created with the `lvipLutTypes::TypeGlobal` parameter in the constructor.
- The 3x3 int32_t array used for the **convolution matrix** is represented by the `LvipConvolutionMatrix` class in the .Net version. All functions for manipulating with the matrix are methods of the `LvipConvolutionMatrix` class.
- The 3x3 int32_t array used for the **color correction matrix** is represented by the `LvipColorCorrectionMatrix` class in the .Net version. All functions for manipulating with the matrix are methods of the `LvipColorCorrectionMatrix` class.

5.4.1. Linking ImgProcLib Class Library with your Application

The ImgProcLib .Net Class Library consists of single file: `lv.simplon.imgproc.net.dll`. This file is an assembly; besides the code, this file contains also the metadata of enumerations and class types, so you do not need to add any other include files to your project. The assembly should be added by the `using` directive to the source code, and also a reference to the library location should be added to the project (Add Reference dialog).

Example in C++:

```
#using <lv.simplon.imgproc.net.dll>
```

Example in C#:

```
using lv.simplon.imgproc.net.dll;
```

Example in Visual Basic .Net:

```
Imports lv.simplon.imgproc.net.dll
```

All classes are placed in the namespace `lv::imgproc`. In C++ avoid using the `lv::imgproc` prefix before each item by specifying the following in the code:

```
using namespace lv::imgproc;
```

The `lv.simplon.imgproc.net.dll` file is installed to `\Bin` folder of Simplon. In order to use it in your application, you must copy it to the folder of your application. When you distribute your application, your setup should install this file (the version used for compilation of your application) to the folder of your application - this conforms to the rules of .Net Framework.

Important note: do not place your application to the `Bin` folder of Simplon. The Simplon setup always places the latest version of `lv.simplon.imgproc.net.dll` to this folder, so the installation of different version of Simplon could make your application non-functional.

The `lv.simplon.imgproc.net.dll` file is a small file, it is a wraparound of the `lv.simplon.imgproc.dll` and thus Simplon itself must be installed as well. The backward compatibility is kept on the level of the `lv.simplon.imgproc.dll`, so it is assured that the class wrapper `lv.simplon.imgproc.net.dll` will work also with newer versions of Simplon.

5.4.2. Error Handling in the .Net Version

When an error occurs, the class throws an exception, which you can catch in standard exception handling way. The raised exceptions are of the `LvipException` class, which is derived from `System::Exception`. If you do not have the exception handling in your code, the error message is displayed in a dialog box and the program is interrupted.

Here is a sample code in C++ how to trap errors:

```
try
{
  m_SourceImage->ApplyLut(nullptr, m_Lut, lvipImgFlags::None);
}
catch (LvipException^ ex)
{
  MessageBox::Show(ex->Message, "Error", MessageBoxButtons::OK,
                   MessageBoxIcon::Exclamation);
}
```

In C#:

```
try
{
  m_SourceImage.ApplyLut(null, m_Lut, lvipImgFlags.None);
}
catch (LvipException ex)
{
  MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK,
                  MessageBoxIcon.Exclamation);
}
```

And in Visual Basic .Net:

```
Try
  m_SourceImage.ApplyLut(Nothing, m_Lut, lvipImgFlags.None)
Catch ex As LvipException
  MessageBox.Show(ex.Message, "Error", _
```

```
                    MessageBoxButtons.OK, MessageBoxIcon.Exclamation)
End Try
```

Note that `ex.Message` contains the *LvipBase::LastStatusMsg*.

5.4.2.1. Error Handling in the .Net Version without Exceptions

By setting the *LvipBase::ThrowErrorEnable* property to `false` you can disable throwing the errors. In such case it is necessary to check the result of each method call.

5.4.3. Classes Hierarchy

All classes are derived from the base `LvipBase` class, which is abstract. The base class provides common properties and methods:

- *LvipBase::ThrowErrorEnable* — see Error Handling.
- *LvipBase::AssemblyVersion* returns the `Version` class containing the current assembly version. Is static, so can be used without need to create the class instance.
- *LvipBase::Log()* — writes a line to the Simplon Log, see Troubleshooting. Is static, so can be used without need to create the class instance.

The following classes are derived from the `LvipBase`:

- `LvipImage`
- `LvipLut`
- `LvipColorCorrectionMatrix`
- `LvipConvolutionMatrix`

5.4.4. The LvipImage Class

Each image handled by the library must be represented by the `LvipImage` class. The image is described by the following items:

- *LvipImage::Width*, *LvipImage::Height* - width and height in pixels. These properties are read-only, use `LvipImage::InitImgInfo()` method to set them.
- *LvipImage::PixelFormat* - pixel format

  This property is read-only, use `LvipImage::InitImgInfo()` method to set the pixel format.
- *LvipImage::Flags* - flags of `lvipImgFlags` type indicating other features and options. Can be combined by the binary `OR` operator.
  - `lvipImgFlags::AttrNotDataOwner` - image data are not owned, belong to another object, so do not deallocate them when the image info is cleared.
  - `lvipImgFlags::AttrColorPlanes` - the image uses 3 separate color planes for RGB
  - `lvipImgFlags::AttrBottomUp` - orientation - first line in the image buffer is the bottom line

  This property is read-write; in case you change the ColorPlanes attribute or the alignment, the current image data buffers are deallocated, if they are owned by this instance.
- *LvipImage::BytesPerPixel* - bytes per pixel, read-only property, calculated form the pixel format.
- *LvipImage::LinePitch* - bytes per line. Read-write property; in case you change it, the current image data buffers are deallocated, if they are owned by this instance.
- *LvipImage::Data* - unmanaged pointer to image data in case color planes are not used. Read-write property.

- *LvipImage::DataR* - unmanaged pointer to the red color plane image data. Read-write property.
- *LvipImage::DataG* - unmanaged pointer to the green color plane image data. Read-write property.
- *LvipImage::DataB* - unmanaged pointer to the blue color plane image data. Read-write property.

The library provides functions for initializing the `LvipImage` structure:

- `LvipImage::InitImgInfo()` - initializes the structure from width, height, pixel format and flags. Besides the image attributes (flags beginning with `lvipImgFlags::Attr`), the flags can contain the `lvipImgFlags::DWordAligned` flag — then the line increment will be aligned to 4 bytes, or `lvipImgFlags::QWordAligned` — then the line increment will be aligned to 8 bytes.
- `LvipImage::CopyFromBmpInfo()` - initializes the structure from Windows `BITMAPINFO` structure
- `LvipImage::CopyToBmpInfo()` - fills the Windows `BITMAPINFO` structure with the current image parameters.
- `LvipImage::AllocateImageData()` - allocates appropriate memory buffer for the image.
- `LvipImage::DeallocateImageData()` - deallocates the allocated memory, if the image buffer is owned by this image info. Note that this method is called automatically in the class destructor and also when any of the image parameters influencing the buffer size is changed, i.e. in `LvipImage::InitImgInfo()` and when the *LvipImage::Flags* or *LvipImage::LinePitch* change.

Contacting Leutron Vision

Headquarters (Switzerland)

Address: Industriestrasse 57, CH-8152, Glattbrugg, Switzerland

Phone: ++41 44 809 88 22

Fax: ++41 44 809 88 29

E-mail (sales): <intsales@leutron.com>, e-mail (support): <intsupport@leutron.com>

Web: www.leutron.com[1]

Germany

Address: Macairestrasse 3, D-78467 Konstanz, Germany

Phone: ++49 7531 59 42 0

Fax: ++49 7531 59 42 99

E-mail (sales): <desales@leutron.com>, e-mail (support): <desupport@leutron.com>

Web: www.leutron.com/de/[2]

Other countries

Customers residing in other countries should have a look at the list of our representatives[3] for a distributor in their country. If no distributor exists in their country, they should contact the headquarter office directly.

Useful links

- To download software, documentation and other stuff, please visit our download area[4].
- Get more information about our support[5]. If you need to return some (defective) material to Leutron Vision, please visit the Return Material Authorization (RMA) page[6].
- Get information about prices and ordering[7].
- Find out detailed information about our hardware and software product range[8].

[1] http://www.leutron.com/
[2] http://www.leutron.com/de/
[3] http://www.leutron.com/sales-contact/
[4] http://www.leutron.com/support-downloads/download-area/
[5] http://www.leutron.com/support-downloads/
[6] http://www.leutron.com/rma/client_id.php
[7] http://www.leutron.com/price-request/
[8] http://www.leutron.com/products-machine-vision-image-acquisition/